

Further Programming for the BBC Micro

Alan Thomas

SHIVA's
friendly
micro
series





Contents

Introduction	1
1 Computing I	3
2 BASIC I	5
3 Graphics I	9
4 Colour	15
5 PRINT I	21
6 Strings I	24
7 INPUT I	30
8 Debugging I	34
9 Sound	37
10 BASIC II	39
11 Graphics II	41
12 Strings II	44
13 PRINT II	50
14 INPUT II	53
15 Debugging II	55
16 Computing II	57
17 BASIC III	61
18 PRINT III	63
19 BASIC IV	65
20 Strings III	70
21 PRINT IV	74
22 BASIC V	76
23 Computing III	80
24 BASIC VI	84
25 PRINT V	85
Additional Programs	87
Program Index	99

Introduction

So you have a BBC computer. Perhaps it is your own one, or it belongs to the school or office. It may be the first computer you've ever touched, or maybe you've got a ZX81 or have used a big mainframe. A new world awaits you when you switch on. This computer is probably more impressive than anything you have seen before. It is an extremely complex and sophisticated piece of machinery, and it will take many hours to master.

This book, although not elementary, merely scratches the surface. It is not:

1. An introduction to BASIC.
2. A User Manual.
3. A copy of the Radio Times.

Its aim is to illustrate by examples the special features of the BBC micro and how to get the best out of it. It helps if you have a little familiarity with BASIC, and you should certainly have the *User Manual* close by. All programs run on the Model A version and have been written so that:

1. Each program can be listed in MODE 7 on the screen without scrolling.
2. Each line of program uses only one screen line with no wrap-around.

Because of these self-imposed restrictions, some of the listings are a little terse. Some of them will run and run for ever. Hit the ESCAPE key if you have to, or better still write in your own method of halting them. You should think of these programs as both examples and as foundations on which you can build.

Here is YOUR program:

```
10 REPEAT
20 Read any program
30 Type it in
40 RUN it
50 Try the suggested alterations
60 Try your own ideas
70 UNTIL fed-up
```

Should you get an ERROR message at any time, don't worry that's life. It simply means that the computer cannot understand what you are trying to do. Because it doesn't think, certainly like you can, it is helpless unless every instruction is crystal clear. It doesn't mind though. Mistakes and errors do it no harm.

Here—try a few to start with:

PRINT 2 + 3

Mistake

print 2 + 3

Mistake

PRINT 1 / 0

Division by zero

PRINT SQR (-1)

-ve root

PRINT ASN (9)

-ve root

PRINT EXP (100)

Exp range

PRINT HALLO

No such variable

Note: A box ☐ is used to represent a space when one is not otherwise obviously needed. If more than five spaces are needed the number required is indicated in square brackets.

1 Computing I

Any computer can calculate directly when asked, just like a pocket calculator. The instruction `PRINT 2 + 4 * 5 / 7` tells your computer to do the calculation and print the result, while `PRINT 10 * 10 * PI` gives the area of a circle of radius 10.

Use of variables enables us to make more complicated calculations. Try this:

```
RADIUS = 10: PRINT 2 * PI * RADIUS: PRINT PI * RADIUS * RADIUS
```

More complicated lines of instruction can be given to the computer, such as:

```
FOR R = 5 TO 10: PRINT R, 2 * PI * R: NEXT
```

Here, the colon indicates to the computer that more than one instruction has been typed on the line.

Normally though, a computer RUNs programs. In BASIC this means giving the computer a list of numbered instructions which it stores in its memory. It doesn't do anything about them until told to with the command RUN. Hopefully, the computer will then follow your instructions as you intended.

Right, so you switch on the power and want to run a program. First you must decide on which MODE to use. By default, you will be in MODE 7. The choice of MODE will depend on what you want your program to do. A rough guide to the choices for a particular MODE are as follows:

MODE 7	TELETEXT characters
MODE 6	ASCII characters
MODE 5	4 colour graphics
MODE 4	Finer graphics than MODE 5
MODE 3	80 characters per line text
MODE 2	16 colour graphics
MODE 1	Mixed text and graphics
MODE 0	Very fine graphics

So MODES 3, 6 and 7 are the natural choices for text and calculations, the others offering a variety of choices of resolution and colour.

Having chosen the MODE, you now have to choose names for your variables. The choice is yours, subject to the conditions laid down in the *Manual*. It may be easier and more meaningful to write statements like:

```
WAGES = HOURS * RATEPERHOUR - DEDUCTIONS
```

rather than `W = H * R - D`.

In a program with many different variables, it is useful to be able to give each variable a meaningful distinguishing name, like numbers on the backs of football players. Tennis players need no labels, and programs with a few variables are usually just as clear with simple one or two character names. In this book short names have been used to keep program listings short.

The computer recognizes three different sorts of variables and stores and manipulates them in different ways. Most common are the floating point (or FP-) numbers, the ordinary numbers we meet in everyday measurements and calculations such as 5.3, PI, $1/7$, $\text{SQR}(2)$. The absolute value of such a variable must lie between $2 \uparrow -128 \approx 2.9 \times 10 \uparrow -39$ and $2 \uparrow 127 \approx 1.7 \times 10 \uparrow 38$ otherwise the computer will think it is “zero” or “too big”. When we count things, we use integers. To indicate that the variable is to be treated as an integer, place the % symbol at the end of its name. For example SCORE% is an integer variable, and so are A%, fred% and PER%, while A, fred and PER are different FP-numbers.

Integer variables must have an absolute value less than $2 \uparrow 31$. Since whole numbers can be considered as FP-numbers, is there any reason to make a distinction? YES! The simple reason is that the computer stores and calculates with integers in a much neater and faster way than with corresponding FP-numbers. The difference in time taken for the same calculation is quite considerable (see TIMETEST) so that one should use integer variables when possible, especially for controlling FOR NEXT loops.

The third type of variable is the string variable consisting of a string of characters. These variables are recognized by the symbol “\$” (read “string” not “dollar”) at the end of their names.

2 BASIC I

The difference between a computer and a calculator is the ability to make decisions, usually by means of "IF (some condition) THEN (some action)". With BBC BASIC you can also specify an alternative action should the condition fail. For example:

```
IF X = 0 THEN PRINT "ZERO" ELSE PRINT "NOT ZERO"
```

Toss a coin 400 times and see if it's:

HEADS OR TAILS

```
10 FOR I = 1 TO 400
20 R = RND (2)
30 IF R = 1 PRINT "H"; ELSE PRINT "T";
40 NEXT
50 PRINT
```

Program note

RND (N) gives a random number between 1 and N, so in this case R is 1 or 2.

*Here's an amusing number problem. Start with some integer.
If it's even divide by 2 otherwise multiply by 3 and add 1.
Now repeat this process and eventually you will reach 1.*

BACK TO ONE

```
10 INPUT N
20 C = 0
```

```

30 PRINT N
40 REPEAT
50 C = C + 1
60 IF N MOD 2 = 0 N = N / 2 ELSE N = 3 * N + 1
70 PRINT N;
80 UNTIL N = 1
90 PRINT
100 PRINT C; "□STEPS"

```

You can also keep REPEATing a process UNTIL some condition is met. The simple line:

```
TIME = 0: REPEAT UNTIL TIME > T
```

will produce a pause of T hundredths of a second.

*Throw two dice until you get doubles:
how many throws does it take?*

DOUBLES

```

10 C = 0
20 REPEAT
30 C = C + 1
40 X = RND (6)
50 Y = RND (6)
60 UNTIL X = Y
70 PRINT C

```

Program note

Surround this program with a loop such as:

```

5 FOR I = 1 TO 100
75 NEXT

```

and possibly include:

```
65 PRINT X
```

and keep a record of what doubles you get, and how long it takes.

The mathematical functions INT and MOD prove very useful, where INT (X) returns the integer part of a number, while X MOD N gives the remainder on dividing X by N.

Calculate which day of the year or century with:

DAYNUMBER

```
10 INPUT "DAY □" D, "MONTH □" M, "YEAR □" Y
20 Z = INT (365.25 * (Y - 1901) )
30 IF Y > 2000 THEN Z = Z + 1
40 IF M < 3 X = D + 31 * (M - 1): GOTO 110
50 LEAP = 0
60 IF Y MOD 4 < > 0 THEN 90
70 IF Y MOD 100 < > 0 THEN LEAP = 1
80 IF Y MOD 400 = 0 THEN LEAP = 1
90 X = 28 * (M - 1) + INT (2.601 * (M + 1) )
100 X = X + D - 7 + LEAP
110 PRINT; D, "/", M, "/", Y, "□ is day □"; X;
120 PRINT "□ of the year and" '
130 Z = Z + X
140 PRINT "day □"; Z, "□ of the century"
150 RUN
```

Program notes

1. Lines 60, 70 and 80 deal with leap years.
 2. Line 90 is a neat way to find the number of days in all the previous months.
 3. You could convert this program to work out your biorhythms.
-

Find the date of:

EASTER

```
10 P = 24: Q = 5
20 PRINT "Which year □";
30 INPUT Y
40 PRINT "Easter Sunday □"; Y, "□ is □";
50 A = Y MOD 19
60 B = Y MOD 4
70 C = Y MOD 7
80 D2 = 19 * A + P
90 D = D2 MOD 30
100 E2 = 2 * B + 4 * C + 6 * D + Q
110 E = E2 MOD 7
```

```

120 I = D + E
130 IF I < 34 THEN 160
140 IF I = 35 PRINT "APRIL 19": STOP
150 IF E = 6 PRINT "APRIL 18": STOP
160 IF I < 10 PRINT "MARCH □"; 22 + I: STOP
170 PRINT "APRIL □"; I - 9: STOP

```

Program notes

1. This formula for Easter is due to C.F. Gauss (1777–1855).
2. Several other religious festivals are determined by Easter. For example Pancake Day is always 41 days before Easter. Modify this program to print the dates of these.
3. Embed this program in a loop.
4. The formula is valid until the year 2099. The values of P and Q must be altered otherwise: for $2100 \leq Y < 2200$ you need $P = 24$ and $Q = 6$.

3 Graphics I

For graphics, the choice of MODE affects the fineness of the points plotted (i.e. the resolution) and the choice of colours available. Whichever MODE you use, points on the screen are referred to by a virtual horizontal coordinate from 0 to 1279 and a virtual vertical coordinate from 0 to 1023, with (0, 0) in the bottom left and (1279, 1023) in top right.

There is a graphics cursor which you can't see. Initially this sits at the origin (0, 0), and drawings are made by moving this cursor. The command MOVE X, Y moves the cursor to the point (X, Y). The command DRAW X, Y moves the cursor to (X, Y) and draws a line as it moves.

Draw a:

RECTANGLE

```
10  X = 100: Y = 200
20  A = 600: B = 400
30  MODE 5
40  MOVE X, Y
50  DRAW X + A, Y
60  DRAW X + A, Y + B
70  DRAW X, Y + B
80  DRAW X, Y
```

Program note

The graphics cursor is moved to the bottom left-hand corner of the rectangle in line 40. Lines 50, 60, 70 and 80 draw the bottom, right side, top and left side respectively. If you write the lines 40 and 50 in reverse, then instead of a rectangle, you'll get two triangles meeting at a vertex.

By joining short lines together you can build a convincing curve.

Draw a regular polygon or a

CIRCLE

```
10 N = 6
20 X = 600: Y = 400
30 R = 300
40 T = 0
50 DT = 2 * PI / N
60 MODE 5
70 MOVE X + R, Y
80 FOR I% = 0 TO N
90 DRAW X + R * COS (T), Y + R * SIN (T)
100 T = T + DT
110 NEXT
```

Program notes

1. The centre of the polygon is the point (X, Y) and its radius is R. It has N sides.
2. As N increases, the polygon looks more like a circle. With N = 30 it's a pretty good circle. Try various combinations of X, Y, R and N.
3. PI in line 50 is approximately π , and is a keyword. SIN and COS in line 90 are two of the mathematical functions available in BASIC.
4. Replace line 90 with DRAW X + R * COS (T), Y + S * SIN (T) and type the extra line 35 S = 200.

Try the program again, and then vary R and S. With a large value of N you will get an ellipse. If you don't know about COS and SIN it doesn't matter. Just use them when necessary. I can walk all day but I don't know how my legs work.

If the values of X, Y refer to a point off the screen, then you may still get a part of the circle. The computer knows where the point (X, Y) is, provided X and Y are less than 32767 in absolute value, so it can "MOVE" and "DRAW" to points off the screen.

INTERFERENCE PATTERNS

```
10 MODE 5
20 ST = 20
30 FOR Y = 0 TO 1024 STEP ST
40 MOVE 5000, 512
50 DRAW 0, Y
60 NEXT
```

Program notes

1. The point (5000, 512) mentioned in line 40 is off the screen.
2. ST is the distance between the end points of these lines. Changing ST will change the pattern.

3. The lines your computer draws are really just lots of dots close together. If these lines are horizontal or vertical they look unbroken, but at any other angle they break up into pieces. Lines drawn close together cause interference patterns as in this program.

The origin for the graphics coordinates is easily moved to the centre of the screen by:

```
VDU 29, 640; 512;
```

The comma and semicolons are *vital* in this line, otherwise you'll get no graphics.

By joining points on a circle, you get a

THREAD PATTERN

```
10 R = 512
20 INPUT N
30 DT = 2 * PI / N
40 MODE 5
50 VDU 29, 640; 512;
60 T = 0
70 FOR J% = 0 TO N - 1
80 U = 2 * J% * PI / N
90 X = R * COS (U)
100 Y = R * SIN (U)
110 FOR I% = 0 TO N
120 MOVE X, Y
130 DRAW R * COS (T), R * SIN (T)
140 T = T + DT
150 NEXT: NEXT
```

Program notes

1. You input N in line 20. The computer takes N points round a circle of radius R and joins each to each other.
 2. Try combining this program with CIRCLE to give thread patterns in different parts of the screen.
-

The flight of a drunken fly is an example of a

RANDOM WALK

```
10 D = 20
20 MODE 5
```

```

30  VDU 29, 640; 512;
40  MOVE 0, 0
50  X = 0: Y = 0
60  R = RND (2): S = RND (2)
70  IF R = 1  X = X + D  ELSE X = X - D
80  IF S = 1  Y = Y + D  ELSE Y = Y - D
90  IF POINT (X, Y) = -1 GOTO 40
100 DRAW X, Y
110 GOTO 60

```

Program notes

1. The random movement of the fly is determined in line 60.
2. The distance the fly moves at each stage is D, set in line 10.
3. If the fly leaves the screen, this is detected in line 90, when another fly appears. POINT (X, Y) gives the colour of the point (X, Y) and is set to -1 if this point is off the screen.

A kaleidoscope of straight lines in black and white:

LINESCOPE 1

```

10  MX = 640: MY = 512
20  MODE 5
30  VDU 29, 640; 512;
40  ST = RND (100)
50  X = RND (MX): Y = RND (MY)
60  SW = 0
70  X1 = X: Y1 = Y
80  IF SW = 1 THEN X1 = MX / ST * RND (ST)
90  IF SW = 0 THEN Y1 = MY / ST * RND (ST)
100 MOVE X, Y: DRAW X1, Y1
110 MOVE -X, Y: DRAW -X1, Y1
120 MOVE -X, -Y: DRAW -X1, -Y1
130 MOVE X, -Y: DRAW X1, -Y1
140 X = X1: Y = Y1
150 SW = (SW + 1) MOD 2
160 GOTO 70

```

Program notes

1. A starting point (X, Y) is chosen at random in line 50. The next point is produced in lines 70, 80, 90 by moving alternately horizontally and vertically from the previous point according to the value of SW.
2. The pattern is drawn symmetrically in lines 100 to 130.

Draw a:

FLOWER

```
10 N = 40
20 DT = PI / N
30 A = 512
40 K = 8
50 MODE 5
60 VDU 29, 640, 512;
70 IF K MOD 2 = 0 THEN DT = 2 * DT
80 E = 0
90 FOR I% = 1 TO K
100 T = PI / (2 * K)
110 FOR J% = 1 TO N
120 X = A * COS (K * T) * COS (T + E)
130 Y = A * COS (K * T) * SIN (T + E)
140 DRAW X, Y
150 T = T + DT
160 NEXT
170 E = E + 0.1
180 NEXT
```

Program notes

1. The number of petals is K, set in line 40. The fineness of the drawing is controlled by N, set in line 10. Try changing the values of K and N.
 2. The time taken to complete the drawing depends on whether K is odd or even (see line 70).
 3. A in line 30 is the diameter of the flower and E (lines 80, 170) controls the overlap of the petals. Try changing A at line 175.
-

*A wheel of radius B rolls inside another of radius A.
This program plots the path of an ant sitting on the inner
wheel, distance R from its centre.*

WHEELS WITHIN WHEELS

```
10 A = 512
20 B = 20 + RND (470)
30 R = RND (1) * B
40 D = A - B: E = D / B
50 T = 0
```

```

60 DT = 0.6
70 MODE 5
80 VDU 29, 640; 512;
90 MOVE D + R, 0
100 REPEAT
110 X = D * COS (T) + R * COS (E * T)
120 Y = D * SIN (T) - R * SIN (E * T)
130 DRAW X, Y
140 T = T + DT
150 UNTIL INKEY (0) > -1
160 RUN

```

Program notes

1. Line 20 chooses a random value of B, not too small nor too near to A.
2. Line 150 is very useful in many programs, coupled of course with a suitable REPEAT. It allows the program to continue UNTIL any key is hit.

4 Colour

Each MODE allows us to use a certain number of colours with which to print text and graphics. These colours have default settings, but can be changed as we shall shortly see.

The colour for text, both foreground and background, is controlled from the COLOUR statement. With $N < 128$ the instruction COLOUR N affects the foreground colour, and $N \geq 128$ the background.

A simple pattern using COLOUR.

STARS AND STRIPES

```
10 MODE 5
20 FOR C = 1 TO 4
30 FOR Y = 0 TO 29
40 FOR X = 0 TO 19
50 COLOUR (X + C * Y) MOD 4
60 PRINT TAB (X, Y); "*";
70 NEXT: NEXT: NEXT
```

Program note

In line 60 a star is plotted in various positions in a colour determined by the position and by the loop variable C.

The colour set depends on $N \text{ MOD } 4$ (the number of colours available). So in MODE 5 where there are 4 colours, COLOUR 19 will be the same as COLOUR 3 since $19 \text{ MOD } 4 = 3 \text{ MOD } 4$ and will set the foreground colour, whereas COLOUR 153 will have the same effect as COLOUR 129, since $129 \text{ MOD } 4$ and $153 \text{ MOD } 4$ are the same, and will set the background colour.

Use COLOUR to draw those

CONTOURS

```
10 MODE 5
20 E = 100
30 FOR Y = 0 TO 29
40 FOR X = 0 TO 19
50 Z = X * X + Y * Y - 100
60 Z = ABS (Z)
70 N = INT (4 * Z / E) + 1
80 COLOUR N
90 PRINT "*";
100 NEXT: NEXT
```

Program notes

1. The value of Z depends on the point (X, Y) and is calculated in line 50. In line 90 a star is printed in a colour depending on the size of Z and E.
2. Try different values of E, and different formulae for Z at line 50.
3. Replace line 80 with COLOUR (N MOD 4).
4. Try adding the line 85 IF Z > E THEN COLOUR 0 and running the program with different values of E.

The command GCOL0 does for graphics what COLOUR does for text. Try the following additions to the earlier programs:

1. To CIRCLE add the lines:

```
85 C = (1 + C) MOD 3
86 GCOL0, 1 + C
```

2. To INTERFERENCE PATTERNS add the line:

```
15 GCOL0, 1
```

3. To THREAD PATTERN add the lines:

```
75 C = (1 + C) MOD 3
76 GCOL0, 1 + C
```

4. To LINESCOPE add the lines:

```
75 GCOL0, C
155 C = (1 + C) MOD 4
```

There are four different GCOL commands. Each of these other GCOL commands tests the colour of the point on the screen before plotting, and calculates which colour to use. In particular GCOL 3 has this simple effect. If the screen point is black, plot as you were going to. If it is the same colour as the point you were going to plot, then rub it out. It is this "rubbing out" feature which makes this command useful and effective.

The Fibonacci sequence can be used to produce a pattern with:

SYMMETRY

```
10 R = 23: S = 512 / R
20 G1 = RND (R - 1): G2 = RND (R - 1)
30 F1 = G1: F2 = G2
40 MODE 5
50 GCOL3, 1
60 VDU 29, 640; 512;
70 REPEAT
80 F3 = F1 + F2
90 F1 = F2
100 F2 = F3 MOD R
110 X = S * F1: Y = S * F2
120 MOVE X, Y
130 DRAW X, -Y
140 DRAW -X, -Y
150 DRAW -X, Y
160 DRAW X, Y
170 UNTIL F1 = G1 AND F2 = G2
180 RUN
```

Program notes

1. The Fibonacci sequence MOD R is generated in lines 80 to 100. The value of R has been set in line 10 to 23, a prime number.
 2. Line 170 checks to see if the pattern is about to repeat, in which case it starts a new one.
 3. The distance between the lines of the pattern is S, which is determined in line 10.
 4. Try this program with different values of R, and different DRAW commands in lines 120 to 160. Leave out 150 and 160, for example.
 5. Include a time delay at line 145 to make the pattern change more slowly.
-

WEAVE

```
10 R = 512
20 X = 0: Y = 0
30 MODE 5
40 VDU 29, 640; 512;
50 GCOL 3, 2
60 P = 10 + RND (50)
```

```

70 Q = 10 + RND (50)
80 DP = 2 * R / P: DQ = 2 * R / Q
90 FOR I% = 1 TO P * Q
100 X = X + DP
110 IF ABS (X) > R DP = -DP: X = X + DP
120 Y = Y + DQ
130 IF ABS (Y) > R DQ = -DQ: Y = Y + DQ
140 MOVE X, Y
150 DRAW -X, Y
160 DRAW -X, -Y
170 DRAW X, -Y
180 DRAW X, Y
190 NEXT
200 RUN

```

Program notes

1. This pattern is determined by the values of P and Q which are chosen between 11 and 60 in lines 60, 70. Try inputting P and Q instead, or changing the numbers 10 and 50 in lines 60, 70.
2. Replace 50 with GCOL3, RND (4) and add a time delay to the end of lines 100 and 120.

You will see from the *Manual* that there are 16 colours available. In MODE 2 you can use them all at once, but in other modes fewer colours are available. Each of these 16 colours has a number from 0 to 15. Colours from 0 to 7 are constant (black = 0 and white = 7) while those from 8 to 15 flash between two colours. The command VDU 19 is used to reallocate the colours. The instruction

VDU 19, I, J, 0, 0, 0

has the effect that whenever I is mentioned in a COLOUR or GCOL statement the colour actually used is colour J. The effect of this instruction is immediate: any points or text already on the screen which were printed or plotted under a COLOUR I or GCOL0, I command will instantaneously change to colour J.

A display of the various colour combinations

COLOUR PAIRS

```

10 MODE 5
20 GCOL0, 1
30 COLOUR 1
40 FOR K = 0 TO 2500 STEP 50
50 MOVE K, 0
60 DRAW 0, K
70 NEXT

```

```

80  FOR I = 0 TO 7
90  VDU 19, 0, I, 0, 0, 0
100 FOR J = 0 TO 7
110 VDU 19, 1, J, 0, 0, 0
120 PRINT TAB (0, 5) "Background"; I
130 PRINT ' ' "Foreground"; J
140 TIME = 0: REPEAT UNTIL TIME > 100
150 NEXT
160 NEXT

```

Program notes

1. The graphics and text colours are set to 1 in lines 20 and 30 which by default is red. In lines 90, 110 the background and foreground colours are changed.
2. Line 140 gives a one second pause between the colour changes.
3. When I = J the background and foreground colours are the same, so you see no text nor graphics. You can avoid this with:

```

105 IF I = J THEN 150

```

Notice that the numbers used by the COLOUR command are not necessarily the same numbers that give the logical colour. In MODE 5, COLOUR 2 will be yellow while logical colour 2 is green. If you want green, you must type VDU 19, 2, 2, 0, 0, 0.

For example, in MODE 6 VDU 19, 0, 7, 0, 0, 0 and VDU 19, 1, 0, 0, 0, 0 will interchange the colours black and white. In MODE 5 the same instructions will replace black by white but it is red that changes to black. White is unchanged.

Try adding to INTERFERENCE PATTERNS the lines:

```

70  FOR I = 0 TO 15
80  FOR J = 0 TO 15
90  IF I = J THEN 130
100 VDU 19, 0, I, 0, 0, 0
110 VDU 19, 3, J, 0, 0, 0
120 TIME = 0: REPEAT UNTIL TIME > 200
130 NEXT: NEXT

```

The 8 flashing colours occur in four pairs; there is a black/white and also a white/black, for example. These are different. The flash rate can be changed from its default setting. The first colour flash rate is controlled by *FX 9 and the second colour flash rate by *FX 10.

FLASH

```

10  MODE 5
20  A$ = "HALLO"
30  *FX 9, 20
40  *FX 10, 80

```

50 COLOUR 1: PRINT ' A\$

60 COLOUR 2: PRINT A\$

70 VDU 19, 1, 9, 0, 0, 0

80 VDU 19, 2, 14, 0, 0, 0

Program note

In lines 50 and 60, A\$ is printed in COLOUR 1 and COLOUR 2. These colours, by default red and yellow, are altered in lines 70 and 80 to a pair of flashing colours. The flash rates have been set in lines 30 and 40 to 0.2 ($= 20 / 100$) and 0.8 ($= 80 / 100$) seconds. This means that COLOUR 9 will be red for 0.2 seconds then cyan for 0.8 seconds while COLOUR 14 will be cyan for 0.2 seconds and then red for 0.8 seconds.

5 PRINT I

The instruction PRINT is fundamental. Without it we would see nothing, however complex and sophisticated the program may be. The programs in this book all make particular use of PRINT and its modifications in order to achieve the desired display of the results.

From the point of view of the PRINT statement, the screen is divided into fields which by default have a width of 10 characters. The command PRINT I, J, K will print each of I, J, K in a separate field, justified to the right. The use of a semicolon instead of a comma ignores the fields and prints the following numbers consecutively. Compare:

```
10 FOR I = 1 TO 10
20 PRINT I, I * I, I * I * I
30 NEXT
```

with

```
10 FOR I = 1 TO 10
20 PRINT I; I * I; I * I * I
30 NEXT
```

If a PRINT line ends with a semicolon, then the next PRINT statement will take effect from this point of the line. Otherwise a new line will be started. Compare:

```
10 FOR I = 1 TO 10
20 PRINT I; I * I; I * I * I;
30 NEXT
```

with the two programs above.

Notice that the cursor is still waiting at the end of the last line. To get it back to its usual position, we need the extra instruction line:

```
40 PRINT
```

The field width is set by the resident integer variable @% which by default is 2570. The total effect of @% is complicated, and an explanation is deferred until later. In simple cases it can be used effectively as follows. If we put @% = 5 then the field width is changed to 5. Since in general each variable is printed in a new field you will now get $40/5 = 8$ fields per line in MODE 7 instead of $40/10 = 4$.

Try the following program.

FIELD WIDTH 1

```
10 @ % = 4
20 FOR I = 1 TO 20
30 PRINT RND (999);
40 NEXT
50 PRINT
```

Program note

The loop simply produces a list of numbers with which you can test the format. Run this program with different values of @% at line 10. Try the values 4, 5, 10 and 20 (all of which divide 40, the length of a screen line) and then 3 and 15.

The value of @% persists until you change it, switch off or BREAK. If you change it for effect in one program, don't forget to reset it afterwards.

The fields set by @% can be overridden using TAB commands. The instruction PRINTTAB(N) moves the printing cursor to position N in the line, ready to act on the next PRINT statement. (If N is too big or the cursor is beyond this point on the line already then new lines will be produced as necessary.)

PRINTTAB(12), X will print X in the next field after position 12, while PRINTTAB(12); X will print X left justified at position 12.

New lines can be produced by a PRINT statement with an apostrophe: PRINT ' ' I ' ' ' J will print 2 new lines, I, 3 new lines then J.

You can also move directly to the point with text coordinates (X, Y) by the command PRINTTAB(X, Y). Here X is the horizontal position measured from the left, and Y is the vertical position measured from the top, so that PRINTTAB(0, 0); Z will cause Z to be printed in the top left corner of the screen.

In a ball game, you need to make it

BOUNCE

```
10 MODE 7
20 MX = 39: MY = 23
30 I = 0: J = RND (39)
40 X1 = I: Y1 = J
50 REPEAT
60 I = I MOD (2 * MX)
70 J = J MOD (2 * MY)
80 X2 = MX - ABS (MX - I)
90 Y2 = MY - ABS (MY - J)
100 PRINTTAB(X1, Y1); "□";
110 X1 = X2: Y1 = Y2
120 PRINTTAB(X1, Y1); "O";
```

```
130 I = I + 1: J = J + 1
140 UNTIL INKEY (0) > -1
```

Program notes

1. The ball, in this case the letter “O” (see line 120), bounces off the edge of the screen at the lines $X = 0$, $Y = 0$, $X = MX$ and $Y = MY$, starting at position (I, J) set in line 30.
 2. The ball moves until any key is hit (see line 140).
 3. Try changing the values of MX and MY in line 20.
 4. Why not try to make this into a game?
-

STAR SYMMETRY

```
10 MODE 5
20 X = RND (19)
30 Y = RND (29)
40 COLOUR RND (4)
50 PRINTTAB(X, Y); “*”;
60 PRINTTAB(X, 29 - Y); “*”;
70 PRINTTAB(19 - X, Y); “*”;
80 PRINTTAB(19 - X, 29 - Y); “*”;
90 GOTO 20
```

Program notes

1. A point (X, Y) is generated at random in lines 20, 30, and a colour in line 40.
2. A star is placed symmetrically on the screen in lines 50 to 80.
3. Try replacing the star with some other character.
4. Try changing all the 19s and the 29s to something smaller. Notice the difference between odd and even values of these parameters.

6 Strings I

We communicate in writing using words, numbers and other symbols. Our “alphabet” comprises 26 upper case letters (A to Z), 26 lower case letters (a to z), 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and various punctuation marks and arithmetic symbols. These are collectively called characters, or more precisely graphic characters.

A string is simply a sequence of characters, the number of which is called the length of the string. The start and finish of a string are indicated by quotes so that A\$ = “ABC” is a string of three characters while “ABC□ □” is a string of five characters, the last two being spaces.

Practise for Mastermind: do you know your

CAPITALS

```
10 MODE 7
20 STQ = 140: NQ = 5
30 TIME = 0: REPEAT UNTIL TIME > 150
40 R = STQ + 10 * (RND (NQ) - 1)
50 CLS
60 PRINT ' "What is the capital of'
70 RESTORE R
80 READ Q$, A$
90 PRINTTAB(10); Q$ '
100 INPUT X$
110 IF X$ = A$ PRINT "Right": GOTO 30
120 PRINT "Wrong it's □": A$
130 GOTO 30
140 DATA Bhutan, THIMBU
150 DATA Bulgaria, SOFIA
160 DATA Libya, TRIPOLI
```

Program notes

1. STQ is the line number of the line containing the first question, and NQ is the number of questions. These are set in line 20.
2. Lines 40 and 70 cause the questions to be asked at random.
3. After a couple of goes, you'll be the expert on the capital of Bhutan. Why not add extra DATA statements with all the capitals you can find?
4. This quiz has no scoring, and it will keep asking you the same question over and over again, even if you get it right. Think about how you might get around those problems. The program CARDS at the end of the book might be useful.

Each character has an associated number, called its ASCII code. This number, which lies in the range 0 to 255 can easily be found since ASC(A\$) gives the ASCII code of the first character of the string A\$. For example ASC("A") is 65, and ASC("a") is 97. Note that the ASCII code for a lower case letter is 32 more than that for the corresponding upper case letter. The digits have ASCII codes from 48 to 57, the letters A to Z from 65 to 90 and a to z from 97 to 122. Not all of the 255 possible codes correspond to a graphic character, but some do have useful effects which we shall describe later.

The character corresponding to ASCII code N is CHR\$(N) so that CHR\$(65) is "A". Try the simple program:

```

10 FOR I = 65 TO 80
20 PRINT I; SPC (2); CHR$ (I)
30 PRINT SPC (2); CHR$ (32 + I)
40 NEXT

```

*Been to a party? Test your***REACTION**

```

10 MODE 7
20 DT = 25
30 R = RND (26) + 64
40 PRINT "Watch for "; CHR$ (R)
50 N = 0
60 REPEAT
70 S = RND (26) + 64
80 IF S = R THEN TIME = 0: N = N + 1
90 PRINT CHR$(S); " ";
100 T = TIME
110 REPEAT UNTIL TIME - T > DT
120 UNTIL INKEY (0) > -1
130 IF N = 0 PRINT "ERROR": GOTO 30
140 IF N = 1 THEN 160
150 PRINT "You missed "; N - 1

```

```

160 PRINT "Time = □"; TIME / 100
170 GOTO 30

```

Program notes

1. The message to “watch for” a certain character appears. You must then hit any key when you see that letter. How long it takes you to react will be displayed.
2. Letters appear at intervals of $D * 0.01$ seconds, in this case every quarter of a second.
3. If you change line 120 to

```
120 UNTIL INKEY (0) = R
```

then you have to hit the letter displayed before the timer stops.

Strings containing spaces are useful in “rubbing out”. When printed the spaces in the string will rub out what was on the screen in those places.

Watch that

SHOOTING STAR

```

10 MODE 7
20 A$ = "□*"
30 FOR T = 10 TO 0 STEP -1
40 CLS
50 PRINT "Delay □"; T
60 FOR I = 0 TO 38
70 PRINTTAB (I, 5); A$
80 TIME = 0
90 REPEAT UNTIL TIME > T
100 NEXT
110 PRINT
120 NEXT

```

Program notes

1. A star moves across the screen at various speeds. These speeds are controlled by the size of T which is used to give a delay of $T * 0.01$ seconds between each PRINT.
 2. Notice that A\$ consists of a space and a star. The space is there to rub out the previous star. See what happens if you leave out this space, or replace it by say a “+”.
-

Keep the children quiet with

KNOW YOUR TABLES

```

10 MODE 7
20 W$ = "Wrong—try again"

```

```

30 R$ = "Right [10 spaces]"
40 N = 1 + RND (11): NR = 0
50 CLS: PRINT
60 FOR I = 1 TO 12
70 PRINT I; "□ × □"; N; "□ = □"; I * N
80 NEXT
90 REPEAT
100 R = RND (12)
110 A = R * N
120 PRINTTAB(17, R); "□ □ □ □";
130 INPUT TAB (17, R), X
140 PRINTTAB(21, R);
150 IF X < > A PRINT W$: GOTO 120
160 PRINT R$
170 NR = NR + 1
180 UNTIL NR = 6
190 RUN

```

Program notes

1. This program tests the N times table: N is set in line 40 to be between 2 and 12. In this line NR, the number of right answers, is set to zero.
2. Spaces in R\$ rub out W\$ if necessary.
3. The spaces in line 120 rub out the correct answer, ready for you to have a go. When you get six right, a new table is started (see lines 90 and 180).
4. Modify this program to keep a count of the number of attempts needed for each question.
5. What about rubbing out all the answers instead of just one, to make it a bit more difficult? Just leave off the I * N in line 70.

The string STRING\$ (N, A\$) is simply N copies of A\$ placed one after the other. For example:

```

10 FOR I = 1 TO 10
20 PRINTTAB(20 - I);
30 PRINTSTRING$(2 * I - 1, "*")
40 NEXT

```

displays a triangle,

```

10 FOR I = 1 TO 20
20 PRINTTAB(30 - I); STRING$ (I, "*")
30 NEXT

```

displays another triangle (this time right-angled), and

```

10 FOR I = 1 TO 10
20 PRINTTAB(I);

```

```

30 PRINTSTRING$(10, "*")
40 NEXT

```

displays a parallelogram.

For a car game you might need a winding

ROAD

```

10 MODE 7
20 L = 16: W = 8
30 R = 38 - L - W
40 T = L + R
50 P = 1
60 REPEAT
70 IF P = 1 Z = RND (R) ELSE Z = RND (L)
80 FOR I = L TO L + P * Z STEP P
90 PRINTSTRING$(I, "*") SPC (W);
100 PRINTSTRING$(T - I, "*")
110 NEXT
120 L = L + P * Z
130 R = 38 - L - W
140 P = -P
150 UNTIL INKEY (0) > -1

```

Program notes

1. This road is W characters wide, the left pavement L and the right R characters wide.
2. The shape of the road changes according to Z, which is set in line 70.
3. To slow down the bends, repeat lines 90 and 100 two or three times.

Unless affected by a PRINTTAB command, a string will be printed left justified, starting in the next field. With the printing of text, @% can be used to tabulate strings into suitable columns.

DIARY

```

10 @ % = 4
20 CLS
30 PRINT "DIARY"
40 REPEAT
50 READ A$, B$, C$, D$
60 IF A$ = "Z" THEN STOP

```

```
70 PRINT A$, B$, C$, D$
80 UNTIL A$ = "Z"
90 DATA MON, JUL, 12, OXFORD
100 DATA MON, AUG, 3, YORK
110 DATA SUN, NOV, 4, FOLKESTONE
120 DATA Z, Z, Z, Z
```

Program note

With @% at its default value 2570, the four strings printed in line 70 would be spread out across the line. By reducing @% to 4, the strings are displayed in a more compact form.

7 INPUT I

Some programs run all by themselves, but most need some INPUT from the user. When the computer wants some INPUT, it will display a cryptic "?". This gives the user no clue to or reminder of what is wanted, which can be very confusing if the program requires different inputs at different points. To avoid this, simply precede each request for an INPUT by a PRINTed reminder or question. Fortunately these two lines can be combined. The statement

INPUT "How old are you □", N

is really PRINT "How old are you" followed by INPUT N. If you leave out the comma before N, no question mark will be printed.

Which day of the week were you born? Find out with

DAYFINDER!

```
10 INPUT "DAY", D, "MONTH", M, "YEAR", Y
20 Z = Y - 1: LEAP = 0
30 IF Y MOD 4 < > 0 THEN 60
40 IF Y MOD 100 < > 0 THEN LEAP = 1
50 IF Y MOD 400 = 0 THEN LEAP = 1
60 X = INT (Z/4) - INT (Z/100) + INT (Z/400)
70 X = X + D + Y - 1
80 IF M = 2 THEN X = X + 3
90 IF M > 2 X = X + LEAP + INT (2.61 * (M + 1) )
100 RESTORE
110 FOR I = 1 TO 1 + X MOD 7
120 READ DAYS$
130 NEXT
140 PRINT; D; "/"; M; "/"; Y; "□ is a □":
150 PRINT DAYS$
160 RUN
170 DATA SUN, MON, TUE, WED, THU, FRI, SAT
```

Program notes

1. Lines 30 to 50 test for a leap year, while line 60 works out how many leap years there have been since year 0.
 2. Line 90 corrects for the number of days in the previous months (compare with DAYNUMBER, line 90). This is based on a formula due to Zeller. His formula uses a 2.6, but because of rounding errors this would give the wrong answer, so we must use something a little bigger.
 3. Change line 170 to contain the names of the days of the week, and make the program print a message if it's a leap year.
-

Convert inches to yards, feet and inches with

HOW FAR?

```
10  MODE 7
20  @% = 4
30  REPEAT
40  INPUT ' "Number of inches □", X
50  I = X MOD 12
60  T = X DIV 12
70  F = T MOD 3
80  Y = T DIV 3
90  PRINT ' X; "□ inches make" '
100 PRINTY; "□ yards";
110 PRINTF; "□ feet";
120 PRINTI; "□ inches"
130 PRINT STRING$(39, "_")
140 UNTIL X = 0
```

Program notes

1. @% is set in line 20 to give a compact display.
 2. What is the weird instruction PRINTY in line 100? Simply a condensed PRINT Y. The computer ignores spaces in program lines when it interprets your program. Of course it is clearer to put in spaces from our point of view, but often when you read other people's programs you'll find these condensed words, so it's a good idea to get used to them as soon as possible. The computer knows that PRINTY is not a variable, because you are not allowed to begin variable names with keywords.
 3. Try writing a different version of this program to convert seconds into hours, minutes and seconds; or pounds into tons, cwts, stones and pounds.
-

If it's your birthday, look at

HAPPY BIRTHDAY VERSE

```
10  H$ = "Happy birthday □"
20  T$ = "to you"
```

```

30 INPUT "What is your name ", N$
40 PRINT "H$; T$; ","
50 PRINT H$; T$; ","
60 PRINT H$; "dear "; N$; ","
70 PRINT H$; T$; "."

```

Program notes

1. I slipped this in here, ready to be used later with HAPPY BIRTHDAY TUNE and possibly UPPER TO LOWER.
2. Try a similar program to print out the sort of circular telling you that your lucky number could win you a fantastic prize if only you agree to look at some book on approval. INPUT your name and address, and fit it into a stereotyped letter, but making it look personal.

When it meets the command INPUT N, the computer pauses and waits for you to give it a number. You can go and have a coffee before you type it in if you want. As you type it, you can change it if you make a typing mistake, no action is taken until you hit <RETURN>. The computer then wakes up and carries on with the program. The string of characters you typed before <RETURN> will be processed by the computer according to the type of variable they are going into. For example, if you type:

```
123.309
```

then this will be happily accepted into an FP-number, integer or string variable, even with letter O in the middle. If it is in response to INPUT A\$, then A\$ will be put to "123.309", a string with 7 characters. If the line was INPUT A, then A will be set to 123.3 because the non-digit "O" will halt the FP conversion process. If it was INPUT A%, then A% will be set to 123, because the integer conversion process will stop at the decimal point.

A single character can be read in with GET or GET\$ which need no <RETURN>. While GET takes the ASCII code of the key hit, GET\$ is that character.

```
REPEAT UNTIL GET = 32
```

gives a pause until the space bar is hit, since ASC " " = 32, while

```
REPEAT UNTIL GET$ = "Y"
```

waits for you to type Y; the first letter of Yes in response to a question perhaps.

There is a timed version of GET, called INKEY. The line N = INKEY (120) causes the computer to start a countdown from 120 in hundredths of a second. If you hit a key before the time is up then, as with GET, N will have the ASCII code of the key you hit. If the time runs out then N is set to -1 and the program continues. You were too late, there is nothing you can do now, the computer marches on.

The instruction REPEAT UNTIL INKEY (0) > -1 will cause a pause until any key is hit.

Watch as your

DICE ROLL

```

10 MODE 7
20 REPEAT

```

```

30 PRINTTAB(10, 10); RND (6); RND (6);
40 UNTIL INKEY (0) = 32
50 REPEAT UNTIL GET = 13
60 RUN

```

Program notes

1. Two dice, in the form of two random numbers between 1 and 6, spin rapidly until the space bar is hit (see line 40). Play proceeds when RETURN is hit (see line 50).
2. Notice the difference between lines 40 and 50: the computer passes immediately over line 40 unless a key is touched, whereas it waits indefinitely at line 50.
3. Try writing a version of this program with 3 or 4 dice.

Since the effect of these single-character input commands is immediate, any key touched by mistake could seriously damage your program. Moreover, keys which are hit are stored in a buffer and remembered. The key you touched several minutes ago could be read in the next time a GET is met. One way to avoid this is to flush the buffer with the command *FX15, 0. This erases any old key hits up to that time.

8 Debugging I

Everyone makes mistakes. Mistakes and errors can send your computer chasing up a dead end or round and round in a non-ending loop, or simply producing either nothing at all or a load of gibberish. There are two main types of errors: syntax errors and runtime errors.

SYNTAX ERRORS

A syntax error occurs when a line of program is meaningless to the computer, often due to a typing error. Perhaps you put `PRIMT` instead of `PRINT`, or left out a bracket or comma. On meeting such an error the computer stops running your program and gives you an error message telling you which line it doesn't like. You then have to list that line and examine it to find the error, retype the line and `RUN` again.

RUNTIME ERRORS

These, often very complicated and difficult to find, occur during the run of the program, when some calculation fails even though each individual line of the program is meaningful. For example the number may be too big, or you try to divide by 0, or some variable doesn't exist. You will still get an error message referring to the line of the program where the error was detected, but the cause of the error may be a long way from there. For example the line might be

```
Y = SQR (X * X - 1)
```

with an error message that you are trying to find the square root of a negative number. So you know that $X * X - 1$ is negative, but why is that? The value of X may be the result of many complex calculations. Which of these caused X to take the wrong value can be difficult and time-consuming to trace.

To track down this error and correct it is the process of debugging. There are no simple rules to what you should do. You learn techniques with time and practice. Here are a few important ones.

1. Print the values of relevant variables in the program directly. If necessary, use the computer to perform some informative calculations with these such as:

```
PRINT X, 1 + X * X, Y/X
```

You may get the message "no such variable" showing you the probable cause of the trouble—you've got the names of your variables a bit muddled, or perhaps you've left out some lines of program.

2. If the error is numeric (too big, division by zero) then try a “dry run” by trying to follow through, with pencil and paper (and pocket calculator) the calculations you expect the computer to make, preferably with simple values.
3. Run the program again with some extra PRINT statements at strategic points. If you feel something is going wrong with the variable I between lines 300 and 320 of the program then the extra statements

```
PRINT "LINE 301", I
```

```
PRINT "LINE 311", I
```

will show what happens between these lines. If you want to follow the flow of the program put in statements like

```
411 PRINT "PASSED THROUGH 411 O.K."
```

```
565 PRINT "565 SHOULD NOT GET HERE"
```

4. The fourth of these simple techniques is to use GET at various places to hold up the program at key points, from which if necessary you can <ESCAPE>. If a loop is running through very quickly, put a G = GET in the loop so that it proceeds only when you hit a key, i.e. at your pace. If using this you think you have detected the occurrence of the error, hit <ESCAPE> to suspend the program in that state, and then try the methods of the previous three paragraphs.
Rounding errors arise very easily and can cause unsuspected difficulties.

ROUNDING ERRORS: EXAMPLE 1

```
10 FOR X = 1 TO 80
20 Y = X / 10
30 Z = X - 10 * Y
40 IF Z = 0 THEN 60
50 PRINT X; SPC(2); Z
60 NEXT
```

This program shows that $10 * (3 / 10)$ is not quite equal to 3.

ROUNDING ERRORS: EXAMPLE 2

```
10 FOR N = 1 TO 20
20 D = 2 / N
30 FOR I = -1 TO 1 STEP D
40 IF I = 1 PRINT N
50 NEXT
60 NEXT
```

This example shows that adding 0.1 on to -1 twenty times does not give 1. As N varies in the outer loop, the value of N is printed only if the inner loop reaches 1. Values of N that are not printed are those which produce rounding errors.

ROUNDING ERRORS: EXAMPLE 3

```
10  FOR X = 1 TO 50
20  Y = SQR (X)
30  Z = Y * Y - X
40  IF Z < > 0 PRINT X, SPC (2); Z
50  NEXT
```

Notice that:

1. $\text{SQR}(6) * \text{SQR}(6)$ is not equal to 6.
2. Sometimes Z is positive, and sometimes it is negative.

9 Sound

The sound capabilities of this computer, like the graphics, deserve a book to themselves. We shall describe the simplest form of sound. There are 4 sound channels. Channels 1, 2 and 3 produce tones while channel 0 produces sound effects. On each channel you can select the volume, pitch and duration of the sound. The volume can be any integer from -15 (loud) to 0 (silent), and the duration any integer from 0 to 255 which determines the duration in units of 0.05 secs. Pitch is set by a number from 0 to 255, each unit being $\frac{1}{4}$ semitone (equitempered) so that 48 units make one octave and there is a range of just over 5 octaves. Middle C is obtained with $P = 53$. The pitch of the sound produced by channel 0 depends on the sound being produced by channel 1.

Each channel stacks the sounds it is given in a queue of length at most 4. This queue can be flushed by adding 16 to the first parameter of SOUND, so that the next note is played immediately, rather than waiting its turn at the end of the queue.

SIREN

```
10  A = 100: B = 135
20  S = 1
30  V = 1
40  INCV = 1
50  FOR I = A TO B STEP S
60  SOUND 1, -V, I, 1
70  NEXT
80  V = V + INCV
90  IF ABS(V - 8) = 7 INCV = -INCV
100 C = A: A = B: B = C
110 S = -S
120 GOTO 50
```

Program notes

1. The pitch I and volume -V of the note are varied up and down in two different ways: the pitch in a FOR NEXT loop, and the volume by adding on the increment INCV which is either 1 or -1.

2. Replace line 60 with:

60 SOUND 17, -V, I, 1

to flush the queue and produce a rapidly varying siren.

Countdown to

BLASTOFF

```
10 CLS
20 TIME = 0
30 FOR I = 10 TO 1 STEP -1
40 PRINTTAB(5, 10); I; SPC (2)
50 SOUND 1, -5, 0, 4
60 IF I = 5 PRINT "IGNITION"
70 REPEAT UNTIL TIME > 100 * (11 - I)
80 NEXT
90 PRINTTAB(5, 10); "ZERO"
100 PRINT "BLAST-OFF"
110 FOR I = 1 TO 15
120 SOUND 0, -I, 4, 20
130 NEXT
140 SOUND 0, -15, 4, 100
150 FOR I = 15 TO 0 STEP -1
160 SOUND 0, -I, 4, 5
170 NEXT
```

10 BASIC II

Leave out this section if you are not interested in mathematics.

PI is a keyword, and takes the value 3.14159265 which approximates π to 8 decimal places. (Compare this with the other approximations of 22/7 and 355/113.) All trigonometric functions expect their angles to be measured in radians, one radian being $\pi/180 \approx 0.017$ degrees, since a right angle is $\pi/2$ radians. The functions RAD and DEG allow easy change between radians and degrees if required.

The inverse trigonometric function ACS (usually called arccos or $\cos^{-1}$) gives values in the range 0 to PI whereas ASN and ATN (arcsin and arctan) give their values in the range $-\pi/2$ to $\pi/2$. An attempt to find ASN or ACS of a number outside the range -1 to 1 will give an error message.

When calculating SIN(T), the computer actually calculates SIN of some other angle which differs from T by integer multiples of $2 * \pi$, making its calculations easier. If T is large, then this simplification could cause rounding errors and produce an inaccurate value. If this is likely then you will get an error message "Accuracy lost". This only applies if the absolute value of T exceeds $2 \uparrow 24 \approx 80000000$.

Other standard mathematical functions available are the exponential EXP and its inverse LN (log to base e), and log to base 10: LOG.

SPIRAL

```
10 N = 10
20 R = 600
30 A = 0.07
40 F = EXP (- A * PI / 2)
50 C = 1
60 DT = 2 * PI / N
70 MODE 5
80 VDU 29, 640; 512;
90 REPEAT
100 T = 0
110 GCOL 0, C
120 MOVE R, 0
130 REPEAT
```

```

140 T = T + DT
150 S = R * EXP (-A * T)
160 DRAW S * COS (T), S * SIN (T)
170 UNTIL S < 50
180 R = R * F
190 C = C + 1: IF C = 4 THEN C = 1
200 UNTIL INKEY (0) > -1

```

Program notes

1. The size of the first spiral is determined by R, which is set in line 20. The smoothness of the curve is controlled by N; a larger value of N gives a smoother drawing, but it takes longer.
2. Subsequent spirals are smaller, according to the value of F (see line 180). The value of F set in line 40 has been chosen so that the fifth spiral overwrites the first, and so on.
3. Try different values of R, F and A.

II Graphics II

MOVE and DRAW are just two of the many available PLOT options, while others are planned to be available in the near future.

The instruction PLOT needs three parameters as in PLOT N, X, Y. Whatever the value of N the cursor, as with MOVE and DRAW, moves to a point specified by the values of X and Y. What it does as it moves is determined by the value of N. Moves can be absolute or relative depending on whether $N \text{ MOD } 8$ is 4, 5, 6, 7 or 0, 1, 2, 3. With $N \text{ MOD } 8 = 4, 5, 6$ or 7 the cursor moves to the point with coordinates (X, Y) measured from the graphics origin. But with a relative move, when $N \text{ MOD } 8 = 0, 1, 2$ or 3 the final point for the cursor has coordinates (X, Y) relative to its previous point. For example if you type `MODE 5: MOVE 640, 512` then the cursor is in the centre of the screen. The command `PLOT 5, 100, 100` draws a line to the point (100, 100), being an absolute instruction, whereas `PLOT 1, 100, 100` draws a line to (740, 612), being relative. The end point of this plot instruction has coordinates (100, 100) measured from (640, 512). PLOT 4 is equivalent to MOVE and PLOT 5 to DRAW. Their relative versions are PLOT 0 (relative move) and PLOT 1 (relative draw).

We know that the DRAW command draws a line in the colour dictated by the current GCOL statement. The versions PLOT 6 and PLOT 7 ignore the GCOL command and plot points in the background colour (rubbing out any points in the way) or in the logical inverse colour, respectively.

So the commands PLOT N with $N = 1, 2, 3, 5, 6, 7$ all draw lines either with relative or absolute coordinates, in a suitable colour, whereas PLOT 0 and PLOT 4 are moves without drawing. Add on 16 to any of these PLOT numbers and you'll get a dotted line instead of a continuous line.

For example, replace DRAW in any of the earlier programs with PLOT 21, and see what happens.

Add on 64 to these PLOT numbers, i.e. PLOT N with N between 64 and 71, will cause the cursor to move, not drawing a line, but placing a single dot at its end point.

Each graphics mode has its resolution, and point size. For example MODE 5 has a resolution of 160×256 . This means it can plot 160 points horizontally, and 256 vertically. Since there are nominally 1280 units across the screen, each point occupies $1280/160 = 8$ horizontal vertical units, and $1024/256 = 4$ units.

Suppose that having drawn a vertical line up the screen you want to draw another beside it. The next program draws a succession of vertical lines, each separated from the next by a nominal distance of ST.

LINEWIDTH

10 @ % = 3

20 MODE 5

```

30 FOR ST = 4 TO 20
40 TIME = 0
50 PRINTTAB(15, 10); ST
60 FOR X = 80 TO 800 STEP ST
70 MOVE X, 0
80 DRAW X, 1024
90 NEXT
100 REPEAT UNTIL TIME > 200
110 CLG
120 NEXT

```

Program notes

1. With $ST = 8$ the lines plotted fit together precisely to give the appearance of a solid block. Try plotting adjacent lines in different colours by adding the lines

```

65 GCOL0, 1 + C
85 C = (1 + C) MOD 2

```

2. With $ST = 16$ there is a gap (or a black line if you like) of one line width between each pair of lines.

Similar thoughts are necessary if you want to draw a vertical line just touching, but not crossing, a horizontal line. Consider for example the program:

```

10 MODE 5
20 Y = 600
30 GCOL0, 3
40 MOVE 0, Y
50 DRAW 1280, Y
60 GCOL0, 1
70 MOVE 640, 0
80 DRAW 640, Y - 4

```

The value $Y - 4$ in line 80 ensures that the vertical yellow line just touches the horizontal red one.

With N between 80 and 87, the command `PLOT N` will fill in triangles. Since any shape you can draw with straight lines can be covered with triangles, this command allows you to fill in areas with colour. Consider for example `PLOT 85, X, Y`. Since $85 \bmod 8$ is 5 this is an absolute move, so the cursor moves to the point (X, Y) measured from the origin. The computer will now fill in the triangle whose three vertices are the point (X, Y) , and the two previous points to which the cursor moved. `PLOT 86` and `PLOT 87` do the same, but the colour effect is as for `PLOT 6` and `PLOT 7`.

Before writing that chess program, you'll need a

CHESSBOARD

```
10  A = 100
20  X = 350: Y = 20
30  MODE 5
40  FOR I = 1 TO 8
50  FOR J = 1 TO 8
60  IF (I + J) MOD 2 = 1 THEN 130
70  P = X + A * (I - 1)
80  Q = Y + A * (J - 1)
90  MOVE P, Q
100 PLOT 1, A, 0
110 PLOT 81, -A, A
120 PLOT 81, A, 0
130 NEXT
140 NEXT
```

Program notes

1. Lines 90 to 120 fill in a square with two triangles. Notice the order in which the corners of the square are taken.
2. The squares of the board are A units across, and the bottom left-hand corner is at (X, Y).
3. Notice the use of relative PLOTS in lines 100, 110 and 120, where the corners of a square are described relative to one corner.
4. This board needs a fancy border. Try and add one made of thin solid rectangles.

12 Strings II

String variables can be tested for equality: IF A\$ = "YES" THEN STOP, but they can also be compared lexicographically so that A\$ < B\$ is TRUE if A\$ would appear before B\$ in a dictionary. More precisely the two strings are compared character by character from the left until they differ. The strings are ordered according to the ASCII code of the first character in which they differ.

For example, CAST < CAT is TRUE because ASC"S" is less than ASC"T". Similarly CAT < CaT is TRUE because ASC"A" is less than ASC"a". In the ASCII ordering, numbers come before the upper case letters which come before the lower case letters. The space character has ASCII code 32 and is the first graphic character, so that "CAT □ AND DOG" is less than "CATANDDOG", and "CAT1" is before "CAT2" and before "CATS".

New strings can be built out of old ones. If A\$ and B\$ are two strings then A\$ + B\$ is the string obtained by writing B\$ immediately after A\$, so that "HALLO" + "TO YOU" is "HALLOTOYOU". Notice that LEN (A\$ + B\$) = LEN (A\$) + LEN (B\$) and that A\$ + B\$ is usually not the same as B\$ + A\$.

There is no minus for strings, but in its place are the three string functions LEFT\$, RIGHT\$ and MID\$, each of which cuts out a piece of the string to which it is applied. For example, if day\$ = "SunMonTueWedThuFriSat" then MID\$(day\$, 3 * (i - 1) + 1, 3) will be:

"Sun" for i = 1, "Mon" for i = 2 and so on.

Deal yourself

ONE CARD

```
10 A$ = "diamondsclubs □ □ □"
20 A$ = A$ + "hearts □ □ spades □ □"
30 MODE 7
130 B% = RND (52)
140 S% = B% MOD 4
150 F% = 1 + B% MOD 13
160 PRINT F%; "□ of □";
170 PRINT MID$(A$, 8 * S% + 1, 8)
```

Program notes

1. The string A\$ of length 32 contains the names of the four suits, each occupying 8 characters. A random number B% between 1 and 52 is taken MOD 4 and MOD 13 in lines 140, 150 to produce the suit and face value of the card.
 2. Modify this program to print Jack, Queen, King or Ace if F% takes certain values.
 3. If you embed this program in a loop to try and get a hand of cards, then you must take care not to deal yourself the same card twice. One way of doing this is to keep a record of the values of B% as they occur. The program CARDS does this, and the lines here have been numbered to be compatible with that program.
-

To check if you have permission to use the program, type in your

PASSWORD

```
10 INPUT "Password □", A$
20 L = LEN A$
30 X = 0: F = 1
40 FOR I = L TO 1 STEP -1
50 E = ASC (RIGHT$(A$, I)) - 64
60 X = X + F * E
70 R = 3 * F
80 NEXT
90 IF X <> 73497 PRINT "INVALID": STOP
100 PRINT "Proceed"
```

Program notes

1. The valid password is COMPUTeR. The string A\$ is converted to an integer X in the loop 40 to 80. This number then determines what happens next.
 2. In a short simple program it would be easy to get around any password check and on to the main program. With a big business computer this would be made as difficult as possible to prevent unauthorized use. It is possible for you to combine a password and coding program (such as CODE for example) so that the program recorded on a cassette is meaningless without knowledge of your password.
-

To protect your precious programs, why not use a

CODE

```
10 MODE 7
20 INPUT "Code number □", N
30 N = N MOD 26: IF N < 0 THEN N = 26 - N
40 PRINT "Type C-to code"
```

```

50 PRINT "Type D-to decode"
60 G$ = GET$: PRINT G$
70 N = N MOD 26
80 PRINT "Message to be ";
90 IF G$ = "D" PRINT "de"; : N = 26 - N
100 PRINT "coded"
110 INPUT M$
120 L = LEN (M$)
130 B$ = ""
140 FOR I = L TO 1 STEP -1
150 C = ASC (RIGHT$ (M$, I)) - 65
160 IF C < 0 OR C > 25 C = C + 65: GOTO 180
170 C = (C + N) MOD 26 + 65
180 B$ = B$ + CHR$ (C)
190 NEXT
200 PRINT B$

```

Program notes

1. The number N (lines 20, 30) is used to code or decode the message.
2. A more complicated code, very difficult to crack, could be constructed as follows. Instead of using the same value of N throughout the loop 140 to 190, take N to be RND (26) and use different numbers each time. To make sure you know which random numbers are being used, "seed" the random number generator to start with, by means of the code number you give the program; add in the line:

```
25 X = RND (-N)
```

and use an eight digit code number.

When you type in a number in response to an INPUT or a GET statement, there is always a possibility of touching the wrong key, often O instead of 0. One way to trap such mistakes is to take each input in character form, and convert it to a number. For example, suppose you have N = GET in the program, and you mean to type in a number. Then N should be between 48 (= ASC("0")) and 57 (= ASC("9")), so N - 48 will give the value of any digit 0 to 9. If N - 48 is less than 0 or greater than 9 then the key hit was not a digit, and the program could be designed to take corrective action, if so desired.

This is simple for single-digit input. For longer strings of digits use VAL, since VAL (A\$) is the numeric value which A\$ represents. For example if A\$ = "12.34000", a string of length 8, then VAL (A\$) will be the number 12.34. If B\$ = "123O.456" then VAL (B\$) will be 123, because the conversion process will halt when the non-numeric character "O" is met. The opposite of VAL is STR\$ which converts a number to a string of digits. This can be useful in printing numbers in a special format, because strings are printed left justified while numbers are right justified.

Think of a number, then find its

FACTORIZATION

```
10 MODE 7
20 INPUT ' ', N%
30 IF N% < 2 THEN STOP
40 M% = N%
50 PRINT N%; "□ = □";
60 IF M% MOD 2 < > 0 THEN 80
70 PRINT "2."; : M% = M% / 2: GOTO 60
80 K% = 3
90 IF M% MOD K% < > 0 THEN 120
100 PRINTSTR$(K%); ".";
110 M% = M% / K%: GOTO 90
120 K% = K% + 2
130 IF K% * K% <= M% THEN 90
140 IF M% = N% PRINT "PRIME": GOTO 20
150 IF M% > 1 PRINT STR$(M%)
160 PRINT
170 GOTO 20
```

Program notes

1. The number input is copied into M%, which is factorized. Each time a factor is found it is printed (lines 70, 100) and M% is replaced by the corresponding quotient. Line 130 prevents the computer searching fruitlessly for factors bigger than the square root of M%.
2. STR\$ is used in line 100 to produce a display with no spaces.
3. Try this program with some large nine-digit integers and see how long it takes. The numbers 123456789 and 987654321 are quite interesting.
4. You could put this program inside a loop with

```
20 FOR N = 1000001 TO 10001000 STEP 2
170 NEXT
```

to find all primes between 1000001 and 10001000.

More sophisticated than VAL is its big brother EVAL. This acts on a string, but actually EVALuates the string, replacing any variable by its current value. The string to be evaluated must be a meaningful formula, such as A\$ = "A * Y * Y + B * Y + C" where A, B, C and Y are existing variables. The next program asks you to solve an equation like "3x + 4 = 7" and will respond with "Right" if you input "1" or "(7 - 4) / 3" or INT (SQR (2)).

Test your

ALGEBRA

```
10  MODE 7
20  R$ = "Right": W$ = "Wrong"
30  @ % = 4
40  x = RND (20) - 10
50  A = RND (10) - 5
60  IF A = 0 THEN 50
70  B = RND (10) - 5
80  y = A * x + B
90  PRINT ' "If □";
100 IF A < > 1 PRINT A;
110 PRINT "x";
120 IF B = 0 THEN 150
130 IF B > 0 PRINT "+"; ELSE PRINT "-";
140 PRINT STR$ (ABS (B) );
150 PRINT "□ = □"; y
160 INPUT ' "what is x", A$
170 XA = EVAL (A$)
180 IF XA = x PRINT R$ ELSE PRINT W$
190 PRINT "□ x is □"; x
200 PRINT STRING$ (39, "-")
210 GOTO 40
```

Program notes

1. A random question is displayed. The numbers used in the question (x, A and B) are determined in lines 40, 50 and 70. For example line 40 gives x a value between -9 and 10.
2. Usually a positive number would be printed without a preceding "+". Lines 130, 140 put a plus or minus in as necessary.

Finally the command INSTR searches one string to see if it contains another.

A simple data search, looking through a

PHONELIST

```
10  NF$ = "□ is not in the list."
20  INPUT ' ' "Name = □", N$
30  N = LEN (N$)
```

```

40 FOUND = FALSE
50 RESTORE
60 REPEAT
70 READ A$, B$
80 IF LEN (A$) < N THEN 120
90 IF INSTR (A$, N$) = 0 THEN 120
100 PRINT ' "Is it □" A$; SPC (4); B$; "?"
110 IF GET$ = "Y" THEN FOUND = TRUE
120 UNTIL FOUND OR A$ = "ZZZZ"
130 IF NOT FOUND PRINT N$NF$
140 GOTO 20
150 DATA BILL (OFFICE), 347743
160 DATA SWANSEA STD, 0792
170 DATA BILL (HOME), 340989
180 DATA JILL, 206064
190 DATA ZZZZ, 4

```

Program notes

1. The search takes place between lines 60 and 120. Each DATA entry is examined to see if it contains the specified string N\$. Any possible name is printed with a query (line 100).
2. The variable FOUND (line 40) which is TRUE or FALSE is tested in lines 120 and 130. The name "FOUND" has been chosen deliberately to make these lines self-explanatory, as an example of how choice of variable names can improve the readability of a program.

13 PRINT II

The format of printing is controlled by the resident integer variable @ %.

The use of this variable is quite complex, as it controls:

- the field width;
- the manner in which numbers are to be printed;
- the degree of precision to which numbers are printed.

The two common ways of representing a number are decimal and exponential (sometimes called scientific) notation. Consider the fraction $X = 101/8$. In decimal notation this is represented by 12.625. Now it may be that for some reason we want to work to say 2 decimal places. Because X needs 3 decimal places we can only achieve this by rounding. To 2 decimal places X would be written as 12.63 and to 1 decimal place simply as 12.6. To 0 decimal places, it would be rounded up to 13. If $A = X / 10000$ and $B = A / 100$ then to 2 decimal places both A and B would be rounded to 0.00, even though A is 100 times B.

The exponential notation seeks to overcome this difficulty by placing the emphasis on significant figures. In exponential notation, $101/8$ is written as 1.2625 E 1 (where E N means $\times 10^N$), enough powers of 10 being extracted from X to produce a number in the range 1 to 10, and it is this number which contains the significant digits. For example $A = 1.2652 \text{ E } -3$ and $B = 1.2625 \text{ E } -5$. If we want to work to fewer significant figures, then it is this number that we change, also by rounding. For example, to 3 significant figures, X will be written as 1.26 E 1 and to 2 significant figures as 1.3 E 1.

Try the next program with various values of @ %, say between 4 and 20. You need to set @ % before running the program, and you can do that directly: simply type

```
@% = 5 <RETURN>
```

for instance.

```
10 PRINT ' ' @% = □': ~ @ % '
20 PRINTSTRING$ (3, "1234567890")
30 X = 0.0001234
40 Y = X
50 FOR I = 1 TO 15
60 PRINT I, Y, I
70 Y = 10 * Y
80 NEXT
```

Program notes

1. Line 20 gives a header to the table.
2. Replace line 30 with INPUT X\$: X = EVAL (X\$) and run the program with different calculations in X\$.

To select decimal (or F) format for printing, put:

$$@% = 2 * 256 * 256 + a * 256 + b$$

where b is the field width and a is the number of decimal places required. To select exponential (or E) notation, put:

$$@% = 256 * 256 + a * 256 + b$$

where b is the field width, and a is the number of significant figures required.

In both these formats integers will be printed along with the rest so that 200 may be written as 200.00 or 2.0 E 2. There is a third (or G) format, the general format, which provides a useful mixture of the E and F formats and leaves integers as integers. To select this mode put

$$@% = 256 * a + b$$

where b is the field width, and a is set with the following in mind. Suppose X is the variable to be PRINTed. In general format:

1. If X is an integer, it is printed as an integer.
2. If $ABS(X) < 0.1$ then X is printed in E format.
3. If $0.1 < ABS(X) < 1$ then X is printed in F format.
4. If $1 < ABS(X)$ then X is printed in F format as space allows, otherwise E format.

If the number to be printed requires more room than the field allows, then in E format extra fields will be used as necessary, while F format will switch over to G format, so that the number could be printed as an integer or using E format. The next program allows you to test the effect of different formats.

FIELD WIDTH 2

```
10 INPUT "FIELD WIDTH ", F
20 C = 256 * 256
30 C = C + F
40 INPUT "CALCULATE ", A$
50 X = EVAL (A$)
60 FOR I = 1 TO 10
70 @% = 2570
80 PRINT I; SPC (2);
90 @% = C + 256 * I
100 PRINT X;
110 PRINT I
120 NEXT
130 GOTO 10
```

Program notes

1. This program shows the effect of field widths in E format. You input a string in line 40 which is evaluated, and then printed in different formats.
2. To try F format, replace line 20 with:

```
20 C = 2 * 256 * 256.
```
3. To try G format, replace line 20 with:

```
20 C = 0
```

14 INPUT II

If a key is held down, after a small delay the character will repeat in a regular fashion. The delay and repeat rate can easily be changed. The command *FX 11, t sets the delay to $t * 0.01$ seconds, and *FX 12, t sets the repeat rate to $t * 0.01$ seconds. For playing games you might want short delay and fast repeat while if young children are using the keys you might select long delay and no repeat. The editing keys can be brought to life by the command *FX 4, 1. These will return an ASCII code when hit, and can be used in games. Remember though that when you want to use them for editing again you must reassign them by *FX 4, 0.

Use the editing keys to

MOVE A STRING

```
10  MODE 7
20  INPUT ' ', A$
30  *FX 4, 1
40  VDU 23; 8202; 0; 0; 0;
50  X = 19: Y = 12
60  L = LEN (A$)
70  MX = 39 - L
80  S$ = STRING$ (L, "□")
90  PRINTTAB(X, Y); A$
100 REPEAT
110  G = GET - 135
120  X1 = X: Y1 = Y
130  IF G = 1 AND X > 0 THEN X1 = X - 1
140  IF G = 2 AND X < MX THEN X1 = X + 1
150  IF G = 3 AND Y < 23 THEN Y1 = Y + 1
160  IF G = 4 AND Y > 0 THEN Y1 = Y - 1
170  PRINTTAB(X, Y); S$
```

```

180 X = X1: Y = Y1
190 PRINTTAB(X, Y); A$
200 UNTIL G = -103
210 VDU 23; 29194; 0; 0; 0;
220 *FX 4, 0

```

Program notes

1. The editing keys are made accessible to the program in line 30, and are reset in line 220.
2. The flashing cursor is turned off in line 40, and on again in line 210.
3. The ASCII codes returned by the editing keys are 136, 137, 138 and 139. These values, less 135 (line 110), are used to control the movement of the string in lines 130 to 160.

The function keys f0 to f9 can be programmed to contain a string of characters so that when hit they reproduce these characters and the computer thinks you have just typed them. They can be set with useful keywords like RUN, LIST or AUTO, or made to contain strings which you need often. To program f4 for example, type:

```
*KEY 4
```

followed by the string you want it to hold. When you hit <RETURN> this key will contain this string. If you need a return in the string, you must type ' M. For example type:

```
*KEY 0 CLS ' M LIST ' M <RETURN>
```

and when you hit f0, the computer receives the message:

```
CLS
```

```
LIST
```

as if you had typed it directly.

Typing *KEY 0 hallo ' M <RETURN> makes f0 issue the string "hallo", followed by return. If your program then has an INPUT A\$ line and you hit f0 A\$ will be set to "hallo" immediately, making INPUT A\$ behave like a GET\$ since no <RETURN> is needed from you. If you want to watch your time, type

```
* KEY0 PRINT`TIME DIV 100 ' M
```

so that hitting f0 will tell you how many seconds you've been sitting there.

15 Debugging II

When an error is detected by the computer, then by default, it stops executing your program and prints an error message. This routine can be changed with an **ON ERROR** command. With this command you tell the computer what to do if an error occurs. For example:

```
ON ERROR STOP
ON ERROR PRINT "Help": GOTO 100
ON ERROR PRINT I, J, A$
```

Each error has a number, which is stored in **ERR**. The line in which the error was detected is stored in **ERL**, while **REPORT** contains the error message.

ON ERROR EXAMPLE

```
10 ON ERROR GOTO 1000
20 INPUT X
30 Y = X + 1 / X
40 Z = SQR (Y)
50 PRINT Z
60 GOTO 20
1000 PRINT "You silly-billy"
1010 VDU 7
1020 REPORT: PRINT
1030 PRINT "In line □": ERL
1040 IF ERR = 17 PRINT "BREAK TO STOP!!"
1050 GOTO 20
```

Program notes

1. Try this program by inputting some values of **X**, including the values 1, 0 and -1.
2. If you try to get out of this program by hitting the **ESCAPE** key, then you get a message telling you what to do (line 1040) instead.

The effect of an **ON ERROR** lasts till the end of the **RUN** or until cancelled by an **ON ERROR OFF** or superceded by another **ON ERROR** command. This can be useful in

debugging, because you can use the computer to do some of the time-consuming checking when the error occurs, by making it execute a debugging program.

When the command **TRACE ON** is issued, the computer will display, enclosed in brackets, the line number of each line it meets until **TRACE OFF** is met. Although this sounds a useful debugging tool (and it is) it has to be used carefully, otherwise the screen will rapidly fill up with line numbers mixed in with the input/output of the program. It is useful if the program is giving no output and you have no idea what is going on.

TRACE EXAMPLE

```
10 INPUT N
20 J = 2 * N
30 K = 1
40 FOR I = 1 TO N
50 K = K/J
60 I = I - 1
70 NEXT
80 PRINT K
```

Program note

After inputting the value 4 say, this program “hangs”, with nothing happening. If you hit **ESCAPE**, and then **PRINT I, J, K** some values will be displayed, showing that something is happening. If you put **TRACE ON** and run again, then the numbers 60 and 70 fill the screen showing that the program is going back and forth between these lines. This shouldn’t happen: line 60 should have been:

```
60 J = J - 1
```

A sensible way to write programs is in chunks, each chunk performing part of the process, with these chunks fitting together to make the program. Chunks can be tried and tested individually. If you do this, and put the tried and tested chunks at high line numbers with the untested chunks at low line numbers, then **TRACE** can be helpful with large programs because **TRACE N** will only trace lines with line numbers less than N.

16 Computing II

This section is for those of you who want to understand a little of the internal storage of the computer. In order to get maximum efficiency from your computer, it is important to know what takes place when you type in a program. Not the intricacies of microchip technology but the way the computer looks at the problem you've given it. Everything you type in and the calculations made are stored inside the memory in a coded form which the computer easily understands. From the earliest computers, the organization of the memory has been based on the binary or two-state system: a current is on or off, a voltage high or low. The two states are represented by the two binary digits 0 and 1. Each positive integer N has a binary representation $a_n a_{n-1} \dots a_0$ each a_i being 0 or 1, where:

$$N = 2^n \cdot a_n + 2^{n-1} \cdot a_{n-1} + \dots + a_0$$

For example the integer 13 is represented as 1101 in binary, because:

$$13 = 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 0 + 1$$

Convert that integer to

BINARY

```
10 INPUT N%
20 M% = N%
30 B$ = ""
40 REPEAT
50 B$ = STR$(M% MOD 2) + B$
60 M% = M% DIV 2
70 UNTIL M% = 0
80 PRINT N%, "□ = b □"; B$
90 GOTO 10
```

Program note

This program, or a version of it, is used in the programs SUPERSTAR and DESIGN PRINT.

As the size of an integer increases, so does the length of its binary representation making binary unwieldy. People who work with computer memory use octal (base 8) or hexadecimal (base 16) notation. In decimal notation we need the 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 in order to represent a number, and in hexadecimal we need 16. We still use 0 to 9 but now need extra symbols for the numbers ten to fifteen, for which we use A, B, C, D, E and F.

Convert an integer to

HEXADECIMAL

```

10  A$ = "0123456789ABCDEF"
20  INPUT N%
30  M% = N%
40  B$ = " "
50  REPEAT
60  Q% = M% MOD 16: M% = M% DIV 16
70  B$ = MID$(A$, 1 + Q%, 1) + B$
80  UNTIL M% = 0
90  PRINT N%; "□ = &"; B$
100 GOTO 20

```

Program notes

1. The string B\$ contains the hex representation of N%. This string it built one character at a time out of the digits stored in A\$.
 2. Write a program to do the reverse: input a string and convert it to an integer, only accepting hexadecimal digits as valid in the string.
-

To indicate that a string is really a hexadecimal number, it is preceded by the & sign, so that &16 is twenty-two, while 16 is sixteen.

Most micros organize their memory into 8 bit pieces called bytes. Each byte can be considered as a binary number from 00000000 to 11111111 or as a decimal from 0 to 255, or a hexadecimal &0 to &FF.

The memory consists of a certain number of bytes, each byte having its own address which is a number from 0 to 65535, i.e. from &0 to &FFFF. A chunk of &100 (= decimal 256) bytes is called a page. Four pages (1024 bytes) make 1K of memory.

When you type in or load a program, it is stored somewhere in the memory. Each character, except for keywords which are treated differently, occupies one byte and the value of that byte is the ASCII code of that character. For example, if "A% = 3" is in a line of program, then 4 consecutive addresses in the memory will contain 65, 37, 61, 51 corresponding to A, %, =, 3. Each keyword is stored as a single byte, the value being the "token" of that keyword. For example PRINT has token 241, so that the line PRINT A will be stored as 241, 32, 65. Each variable used in the program must have somewhere to store its current value. Each integer is allocated 4 consecutive bytes. If this integer is positive, then these 4 bytes together are the binary representation of that number. This is why integers cannot exceed 2^{31} .

Each FP number occupies 5 consecutive bytes, in a binary exponential form. That means that if x is a positive number, then the computer first expresses x in the form $2^n * y$ where n is an integer and $1/2 < y \leq 1$. For example 7 will become $2^3 * (7/8)$. The value of x is stored as follows: the first byte is 128 + n, and the remaining 4 bytes contain a binary representation of y, taking account of sign. Since the first byte can only hold 255 at maximum, n must be less than 128 which is why 2^{127} is the largest FP number the computer can deal with.

Because it only allows 5 bytes for each FP number, there will be some numbers, many in fact, which it cannot store exactly. Some rounding errors will occur. But this would be true however many bytes you put aside to store each variable. We know that $1/3 = 0.33333 \dots$ is an infinite repeating decimal. Any attempt to force this into a fixed amount of space must incur rounding errors.

Write 1/N% to four hundred decimal places with

RECIPROCAL

```

10  E% = 4: K% = 10 ↑ E%
20  T% = 100
30  INPUT N%
40  IF N% <= 1 THEN STOP
50  PRINT "1/"; N%; "□ = □" ' "0. ";
60  R% = 1
70  FOR I% = 1 TO T%
80  M% = K% * R%
90  A% = M% DIV N%
100 R% = M% MOD N%
110 IF R% = 0 THEN I% = T%
120 A$ = STR$(A%): L = LEN A$
130 PRINT STRING$(E% - L, "0"); A$;
140 NEXT
150 PRINT
160 GOTO 30

```

Program notes

1. The value of $1/N\%$ is printed to $4 * T\%$ decimal places, with repeating zeros suppressed (line 110). Lines 120 and 130 fill in zeros as necessary.
2. Because of the way integers are stored, errors may occur if you try to multiply two five-digit integers together, but not two four-digit integers. This is why $E\%$ is set to 4 and $K\%$ to 10000.
3. Notice the missing space in $THEN I\%$ in line 110.

The number $1/7$ has an infinite repeating decimal expression, and also an infinite repeating binary one. If we put $X = 1/7$ in a program, then X stores a number which is not quite $1/7$, and so $7 * X$ is not quite 1. Compare:

```
PRINT 7 * (1/7)
```

with

```
PRINT 1 - 7 * (1/7)
```

for example

The next program shows exactly how FP numbers are stored.

NUMBER STORAGE

```
10  @ % = 4
20  X = 0
30  INPUT X$
40  X = EVAL (X$)
50  PRINT ' X$; "□ = □"; X
60  FOR I = 3 TO 7
70  PRINT ~ ? (LOMEM + I);
80  NEXT
90  PRINT ' STRING$ (39, "-")
100 GOTO 30
```

Program notes

1. A string to be evaluated is input in line 30, and evaluation takes place in line 40. The loop 60, 70, 80 displays those memory locations that store the variable X.
2. Try and compare the following calculations:
 - (a) $7 * 1/7$ and $7 * (1/7)$
 - (b) $\text{SQR}(6) * \text{SQR}(6)$ and 6
 - (c) 2, 4, 8, 16 and 32
 - (d) 3, 6, 12 and 24
 - (e) PI and $4 * \text{ATN}(1)$
3. The symbol ~ in line 70 causes the following variable to be displayed as a hexadecimal number.
4. LOMEM is the address of the first variable used by the program, in this case X. The question mark (= PEEK) extracts the contents of the relevant memory location.

17 BASIC III

Logical decisions made by the computer control the flow of a program. Appropriate action is taken according to whether some condition is true or false.

The logical functions AND, OR and NOT are self explanatory. The exclusive or EOR (often confused with OR) is not so common. The statement A EOR B is only true if A is true or B is true, but not both. If you interpret TRUE as the binary digit 1 and FALSE as 0 then the values of AND, OR and EOR can be read from the tables below:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

EOR	0	1
0	0	1
1	1	0

Each integer has a binary representation, and the logical functions extend to functions between integers, applying to each corresponding pair of bits in turn.

EOR TABLE

```
10  MODE 7
20  @% = 2
30  PRINT " " "□ □ □!";
40  FOR I = 0 TO 7
50  PRINT I; : NEXT: PRINT
60  PRINT STRING$(10, "—")
70  FOR I = 0 TO 7
80  PRINT I; "□!";
90  FOR J = 0 TO 7
100 N = I EOR J
110 PRINT N;
120 NEXT
130 PRINT
140 NEXT
```

Program notes

1. This program displays in tabular form the function EOR applied to small values of I and J. Line 60 and the exclamation mark in line 30 produce a horizontal line and a vertical line at the edge of the table.
 2. Try a version of this replacing I EOR J with various other logical functions such as:
I AND J, I OR J, I AND NOT J, NOT I EOR NOT J
-

The keywords TRUE and FALSE are integers whose bits are all 1 and all 0 respectively. Because of the way that negative integers are stored, TRUE is actually -1 while FALSE is 0. Any logical statement has the value -1 if TRUE and 0 if FALSE. This value can be used by your program. For example, $X = X + (X > 1)$ will subtract 1 from X if X > 1 is TRUE, otherwise X is unchanged.

Convert a string of letters from

UPPER TO LOWER

```
10 INPUT A$
20 L = LEN A$
30 B$ = LEFT$(A$, 1)
40 FOR I = L - 1 TO 1 STEP -1
50 G = ASCRIGHT$(A$, I) OR 96
60 B$ = B$ + CHR$G
70 NEXT
80 PRINT B$
90 GOTO 10
```

Program notes

1. If N is the ASCII code for some letter, then N OR 96 is the ASCII code for the corresponding lower case letter. The loop 40 to 70 replaces each letter of the string A\$ (except the first) by the equivalent lower case letter.
2. To convert letters to upper case use "AND 95" instead of "OR 96".

18 PRINT III

When printing, the computer knows where the cursor is; its vertical position (measured from the top of the screen) is stored in VPOS and its horizontal position (measured from the left) in POS. It also remembers how many characters, in COUNT, it has printed since the last PRINT. So in MODE 7 for example, POS lies between 0 and 39, VPOS between 0 and 24 while COUNT can be between 0 and 255. These numbers can be used to control the layout of the screen, especially when you are printing lists and strings of varying lengths. For example, you may wish to pause when the screen is nearly full.

VPOS EXAMPLE

```
10 CLS
20 FOR I = 1 TO 50
30 X = RND (10)
40 FOR J = 1 TO X
50 PRINT I, X
60 IF VPOS < 20 THEN 110
70 VDU 7
80 PRINT "Press spacebar to continue"
90 REPEAT UNTIL GET = 32
100 CLS
110 NEXT
120 PRINT
130 NEXT
```

Program note

The display proceeds until 20 lines are on the screen (line 60) when you get a beep and the display halts until you want it to continue. When you do, hit the space bar.

POS EXAMPLE

```
10 FOR I = 1 TO 100
20 X = RND (9)
```

```
30 PRINT STRING$(X, STR$(X));  
40 PRINT "□";  
50 IF POS > 30 THEN PRINT  
60 NEXT
```

Program note

The strings being printed are of random length. To avoid these overflowing on to the next line, a new line is printed (line 50) once the line becomes too long.

You can use POS to make sure that lines don't go off the edge of the screen. The screen will scroll when the print cursor reaches the end of the bottom line. A carefully thought out program should not need to scroll, being designed "pagewise".

The number of characters per line is determined by MODE, but can be reduced if required by WIDTH. For example if you intend to display some text in MODE 5, but wish to experiment with the layout in MODE 7 then WIDTH 20 will limit your line to 20 characters.

19 BASIC IV

A simple array is just a list. The dimension of an array, set by a DIM statement, is the number of things in the list. You can have arrays of any type. For example:

```
DIM A (10) allocates space for an FP array with 10 elements;  
DIM PA% (100) for an integer array with 100 elements;  
DIM NAME$ (20) for a string array with 20 elements.
```

The individual elements of an array AR are referred to by their position in the list, so that AR (1) is the first, AR (2) the second and so on. (Actually there is an AR (0) which you can and should use.) The elements of the array can be input, read in from DATA or calculated in the program.

Arrays can have more than one dimension, but if you want to use them you'll know what they're about. Once set, the dimension of an array cannot be altered in the program, but it can be read in or calculated within the program before being set.

HAPPY BIRTHDAY TUNE

```
10 DIM A (8)  
20 FOR I = 0 TO 7: READ A: A (I) = 4 * A: NEXT  
30 DATA 0, 2, 3.2, 5, 7, 8, 10, 12  
40 READ N$  
50 REPEAT  
60 N = ASC (N$) - 65  
70 IF N > 7 OR N < 1 THEN 130  
80 D = 4 * VAL (RIGHT$ (N$, 1) )  
90 B = 53 + A (N)  
100 IF MID$ (N$, 2, 1) = " " THEN B = B + 48  
110 SOUND 1, -10, B, D  
120 SOUND 1, 0, 0, 1  
130 READ N$  
140 UNTIL N$ = "Z"  
150 DATA G1, G1, A'2, G2, C'2, B'4  
160 DATA G1, G1, A'2, G2, D'2, C'4
```

170 DATA G1, G1, G'2, E'2, C'2, B'2, A'2

180 DATA F'1, F'1, E'2, C'2, D'2, C'4, Z

Program notes

1. The pitch and duration of each note played is stored in a string, held in the DATA statements 160 to 180. Standard letters A to G are used for the notes, with a prime indicating an octave higher. Each string is "decoded" in lines 60 to 100.
 2. The array A stores the 8 notes of a major scale.
 3. The key, in this case C, is set by the number 53 in line 90.
 4. Line 120 gives a slight pause at the end of each note. Try the program with this line left out.
-

Watch the dice scores build up as you throw

TWO DICE

```
10 MODE 7
20 X$ = CHR$(255)
30 DIM A(12)
40 N = 0
50 PRINTTAB(0, 23);
60 PRINT "Two dice thrown";
70 PRINTTAB(16, 23) N; "□ times"
80 FOR I = 2 TO 12
90 PRINTTAB(0, 2 * I - 2); I; CHR$(146);
100 NEXT
110 REPEAT
120 N = N + 1
130 P = RND(6): Q = RND(6): X = P + Q
140 S = A(X) + 1
150 A(X) = S
160 PRINTTAB(S + 4, 2 * X - 2);
170 IF P = Q PRINT "D" ELSE PRINT X$
180 PRINTTAB(16, 23) N
190 UNTIL S > 30
```

Program notes

1. The array A stores the scores, A(N) being the number of times the score has been N (see lines 140, 150).
 2. After each throw, the result is plotted, printing the letter "D" for doubles.
 3. The program stops (see line 190) when one of the scores exceeds 30.
-

CONVERSION

```
10 NS = 5
20 DIM A$(NS), F(NS)
```

```

30  FOR I = 1 TO NS
40  READ A$(I), F(I): NEXT
、 50  DATA INCHES, 36, FEET, 3, YARDS, 1
60  DATA MMS, 91.44, METRES, 0.9144
70  CLS: PRINT " " "Conversion"
80  FOR I = 1 TO NS
90  PRINT I, "— — —"; A$(I): NEXT
100 PRINT " " "□ □ from □";
110 G = VAL (GET$)
120 IF G < 1 OR G > NS THEN 110
130 A$ = A$(G): F1 = F(G)
140 PRINT A$, "□ to □";
150 G = VAL (GET$)
160 IF G < 1 OR G > NS THEN 150
170 B$ = A$(G): F2 = F(G)
180 PRINT B$
190 PRINT "How many □"; A$: INPUT X
200 PRINT ' X; SPC(2); A$, "□ = □";
210 Y = F2 * X/F1: PRINT Y; SPC(2); B$
220 GOTO 190

```

Program notes

1. This program displays a "menu" of options, in this case NS options, with NS set in line 10. The names and conversion factors for each of the options is read from DATA statements in lines 30 to 60.
2. Line 220 sends the program back to deal with another conversion between the same units. If this line were "GOTO 120" then you could change these as well.

Like crosswords? Try

ANAGRAM

```

10  DIM A(20)
20  READ A$: IF A$ = "*" THEN STOP
30  L = LEN A$: B$ = " "
40  FOR I = 1 TO L
50  A(I) = I: NEXT
60  FOR I = L TO 2 STEP -1
70  X = RND(I)
80  T = A(X): A(X) = A(I): A(I) = T
90  NEXT
100 FOR I = 1 TO L
110 B$ = B$ + MID$(A$, A(I), 1)

```

```

120 NEXT
130 PRINT ' B$ '
140 INPUT W$
150 IF W$ = A$ PRINT "Right □" ': GOTO 20
160 PRINT "No □"; A$: GOTO 20
170 DATA OSTRICH, PENGUIN, CINEMA, *

```

Program note

The array A is used to store a permutation of the letters of A\$. This permutation is produced in lines 60 to 90, and the corresponding permuted string, or anagram, is constructed in the lines 100 to 120.

Keep the family busy playing

HANGMAN

```

10 MODE 7
20 C = 0: NL = 11: R = NL + 6: S$ = "□" + CHR$7
30 DIM A (20), B (20)
40 READ A$: L = LEN (A$)
50 FOR I = 1 TO L
60 A (1 + L - I) = ASCRIGHT$ (A$, I)
70 B (I) = 32: NEXT
80 PRINTTAB (2, 11) STRING$ (L, "-")
90 FOR I = 1 TO NL
100 PRINTTAB (30, I + 5) "*": NEXT
110 G = GET: SW = TRUE
120 FOR I = 1 TO L
130 IF B (I) = G THEN 150
140 IF A (I) = G SW = 0: B (I) = G: C = C + 1
150 NEXT
160 PRINTTAB (2, 10);
170 FOR I = 1 TO L
180 PRINT CHR$ (B (I) ); : NEXT: PRINT
190 IF SW R = R - 1: PRINTTAB (30, R) S$
200 IF R = 6 PRINT "YOU LOSE": STOP
210 IF C = L PRINT ' "YOU WIN": STOP
220 GOTO 110
230 DATA HIPPOPOTAMUS

```

Program notes

1. A string read from a DATA statement is stored in the array A by means of its ASCII code, while B initially contains all spaces (ASCII code 32).

2. At each guess, the answer in B is updated in lines 120 to 150. If this letter is absent or has been used already, you lose one of your lives: a star is rubbed out and you get beeped. If present in A\$ it is printed in lines 170, 180.
3. If you add some extra data statements, and change the STOPS to GOTO 20, it looks as if the program will run until out of data. However you would also need to remove line 30 and place it at, say line 15.
4. You could add “; CHR\$(G)” on to the end of line 190 so that the wrong letters are also displayed.

20 Strings III

Although each graphic character has an ASCII code in the range 0 to 255, not every code corresponds to a graphic character. Some of these other characters affect the printing of following characters, and are called control characters. Some of these apply in every mode, while some have very special effects only in MODE 7. These “characters” can be manipulated along with other characters and combined to form strings. It’s just that when PRINTed you don’t get a visible character. For example CHR\$ (7) is the “bell” character. Characters 8, 9, 10 and 11 move the cursor left, right, down and up.

For example, the program:

```
10 FOR I = 65 TO 85
20 PRINT CHR$ (I); CHR$ (10);
30 NEXT
```

displays letters diagonally from top left to bottom right while:

```
10 PRINTTAB (20);
20 FOR I = 65 TO 85
30 PRINT CHR$ (I);
40 PRINT CHR$10CHR$8CHR$8;
50 NEXT
```

displays the same letters diagonally, this time from top right to bottom left.

Character 12 clears the screen, 13 returns the cursor to the start of the line and 30 homes the cursor in the top left corner of the screen. For example, if:

$A\$ = \text{CHR\$ (12)} + \text{CHR\$ (10)} + \text{"ERROR"} + \text{CHR\$ (7)}$

then A\$ is a string of length 8 with 5 graphic characters. If you PRINT A\$, then the effect will be to clear the screen, print “ERROR” one line down and beep. Character 127 deletes as it backspaces, so that PRINT STRING\$ (5, CHR\$ (127)) will move the print cursor back 5 places, rubbing out what was there, unlike PRINT STRING\$ (5, CHR\$ (8)) which moves the cursor back without rubbing out.

Since these characters don’t get printed, to write PRINT CHR\$ (7) is somewhat a misnomer. If you want to put a lot of these characters together, then the resulting line looks frightening:

```
PRINT CHR$ (30); CHR$ (10); CHR$ (10); CHR$ (9); CHR$ (7)
```

The VDU command controls many aspects of printing, both on screen and printer. In its simplest form VDU N is equivalent to PRINTCHR\$ (N) so that VDU 7 will beep. This

command can have any number of parameters, and processes them consecutively so that

VDU 30, 10, 10, 9, 7

is equivalent to the line above.

Test your aim with

DUCK SHOOT

```
10  MODE 7
20  MX = 39: MY = 22: DT = 5: D$ = "□ d"
30  B$ = "□" + CHR$8 + CHR$11 + "b"
40  X = 35: Y = RND (20)
50  PRINTTAB (X, MY); B$
60  I = 0
70  PRINTTAB (I, Y); D$;
80  I = I + 1: IF I = MX THEN RUN
90  TIME = 0: REPEAT UNTIL TIME > DT
100 IF INKEY (0) < > 32 GOTO 70
110 HIT = FALSE
120 K = X - POS + 1: L = MY - Y
130 IF K = L THEN HIT = TRUE
140 N = MY
150 PRINTTAB (I, Y); D$;
160 PRINTTAB (X, N); B$;
170 TIME = 0: REPEAT UNTIL TIME > DT
180 I = I + 1: N = N - 1
190 IF I = X AND HIT THEN VDU 7: RUN
200 IF I = MX OR N = 0 THEN RUN
210 GOTO 150
```

Program notes

1. Duck and bullet are in D\$ and B\$, together with some spaces.
2. DT, set in line 20, controls the speed of flight and of the bullet.
3. When the space bar is hit, the bullet is fired (line 100) and the program immediately calculates to see if the duck will be hit (lines 110 to 130).

In MODE 7 there are control characters which change the colour, and even the shape, of the following characters.

TELETEXT TEST

```
10  MODE 7
20  @% = 4
```

```

30 L = 129
40 M = 135
50 A$ = "ABCdef123*"
60 FOR I = L TO M
70 FOR J = L TO M
80 PRINT I, J;
90 VDU I, 157, J
100 PRINT A$
110 G = GET
120 NEXT: NEXT

```

Program notes

1. The string A\$ (line 50) is printed in various colours against various backgrounds.
2. Replace lines 30 and 40 with

```

30 L = 145
40 M = 151

```

and run it again. This time part of A\$ is replaced by blocks.

3. Line 110 allows you to control the display: hit any key and the next colour is displayed. Compare this with the similar example COLOUR PAIRS which has a built in time delay instead.

First of all characters 129 to 135 change the colour. The effect of these colour characters lasts till the end of the line or until another colour control character is met. Try

A\$ = CHR\$(129) + "HALLO" and B\$ = CHR\$(130) + "□ EVERYONE"

with

```
PRINT A$; PRINT B$ and PRINT B$ PRINT A$
```

Next 136 and 137 turn the flashing on and off. For example, if:

A\$ = CHR\$(136) + CHR\$(129) + "DANGER" + CHR\$(137)

then when you PRINT A\$ you'll get a flashing red warning. Character 141 doubles the size vertically. The first time it's used you get the top half, and the second time the bottom half of the string. If A\$ = "BIG LETTERS" then PRINTCHR\$(141) A\$: PRINTCHR\$(141) A\$ produces the two halves together.

Characters 157 and 156 set the background colour on and off for the current line. On its own, 157 produces a white blank:

```
PRINT CHR$(157) CHR$(129) "RED ON WHITE"
```

Now try:

```
PRINT CHR$(130) CHR$(157) CHR$(129) "What now I ask?" CHR$(156)
```

Here 130 and 157 combine to produce a yellow background for our red text. The 156 switches the background back to black. This line could have been written as:

```
VDU 130, 157: PRINT CHR$(129) "What now I ask?" CHR$(156)
```

When PRINTed, character 152 hides the remainder of the line, so that

```
PRINT TAB(10); CHR$(152)
```

will clear the line except for the first 10 characters.

The characters from 145 to 151 also change the colour just like 129 to 135. In fact their effect on upper case characters is the same. But they have an interesting effect on lower

case characters which are converted into little coloured shapes. These shapes are based on a 3×2 rectangular blank, with 6 squares.

1	2
4	8
16	32

	2
4	
16	

$$2 + 4 + 16 + 32 = 54$$

1	
	8
	32

$$1 + 8 + 32 + 192 = 233$$

Any combination of these 6 squares can be coloured or not to give $2^6 = 64$ possible combinations. These shapes are all obtainable, in the following way. Write in the squares the numbers 1, 2, 4, 8, 16, 32 as shown. Add up the numbers in the shaded squares. If squares 1, 4, 16 are empty and squares 2, 8, 32 are shaded this gives the answer $2 + 8 + 32 = 42$. If this number is less than 32 add on 32 else add on 192, to give you the magic number, say N. If you PRINTCHR\$(N) normally you will probably get a letter or digit, but if you precede this PRINT by one of the characters from 145 to 151 the shape you drew will appear in the appropriate colour. Each lower case letter determines such a shape corresponding to the magic number which is 128 more than its ASCII code. These block shapes, which correspond to ASCII codes from 32 to 63 and from 224 to 255 can be put together in strings, mixed with cursor movements like CHR\$10, to build up larger block shapes. Of course they will only be printed as such when they follow a CHR\$L with L in the range 145 to 151.

Construct your own

TELETEXT SHAPES

```

10 DATA 147
20 DATA 54, 243, 233
30 DATA 8, 8, 8, 8, 10
40 DATA 145
50 DATA 229, 243, 58
60 DATA 11, 0
70 MODE 7
80 READ N: IF N = 0 THEN 110
90 A$ = A$ + CHR$N
100 GOTO 80
110 FOR I = 1 TO 8
120 PRINTTAB (2); A$;
130 FOR J = 1 TO 8
140 PRINT A$;
150 NEXT: PRINT ' '
160 NEXT

```

Program notes

- Lines 10 and 40 set the colours of the two halves of the shape; line 20 gives the top, and line 50 the bottom. Lines 30 and 60 move the cursor to the right position. Try leaving out an 8 or 10 from line 30 or the 11 from line 60.
- Once stored in A\$, i.e. after line 100, this shape can be printed anywhere, any time.

21 PRINT IV

The size of the screen used for text can be set by a VDU 28 command, which requires two pairs of text coordinates, i.e. 4 parameters. These are the coordinates of the bottom left and top right corner of the "text window" you require. For example, if you want to use some of the screen for graphics and some for text, try the following program.

Set a text window above a graphics window.

WINDOW

```
10 MODE 7
20 INPUT "How many text lines □", N
30 Y = 1024 - 32 * N
40 MODE 4
50 PRINT SPC(5); "TEXT AREA"
60 PRINTTAB(5, N); "GRAPHICS AREA"
70 VDU 28, 0, N - 1, 39, 0
80 VDU 24, 0, 0; 1279; Y;
90 MOVE 0, Y
100 DRAW 1279, Y
```

Program notes

1. In MODEs 0 to 5 there are 32 lines of text, so each line uses $1024/32 = 32$ graphics units. This explains the constant 32 in line 30.
 2. This program sets an N-line text area above the remaining graphics area, and draws a line across the screen separating the two areas.
-

Set a graphics window above a text window.

ANOTHER WINDOW

```
10 MODE 7
20 INPUT "How many text lines □", N
30 MODE 4
```

```

40 PRINT "Graphics area"
50 VDU 28, 0, 31, 39, 32 - N
60 VDU 24, 0, 32 * N; 1279; 1023;
70 VDU 29, 0, 32 * N;
80 MOVE 0, 0
90 DRAW 1279, 0
100 PRINT "Text Area"

```

Program notes

1. This time the text area is below the graphics area, the common way to display mixed text and graphics.
2. In line 70 the graphics origin is moved to the bottom left-hand corner of the graphics area.

Each time you use a VDU 28 command it overrides the previous setting. To restore the default setting, type VDU 26.

In any graphic mode, the text and graphics cursors can be linked together by VDU 5. Once issued the PRINTing will always take place from where the graphics cursor is, until cancelled by BREAK or VDU 4. There is no longer a "next line" and TAB has no effect.

Display your

CHEMICAL FORMULAE

```

10 MODE 4
20 VDU 5
30 READ A$: IF A$ = "*" VDU 4: STOP
40 CLG: MOVE 100, 900
50 FOR I = 1 TO LEN A$
60 B$ = MID$ (A$, I, 1)
70 PLOT 0, 0, -16 * SGN (VALB$)
80 PRINT B$;
90 PLOT 0, 0, 16 * SGN (VALB$)
100 NEXT
110 REPEAT UNTIL GET = 32
120 GOTO 30
130 DATA H2SO4, H2O, CuSO4, Ca(NO3)2, *

```

Program notes

1. Text and graphics cursors are joined in line 20, and are separated at the end in line 30.
2. In line 70, VALB\$ will be zero if B\$ is not a digit, in which case B\$ is printed on the line. Otherwise B\$ will be printed half a line below. Line 100 resets the print cursor on to the line.

22 BASIC V

Often within a program you need to make the same sort of calculation or go through the same process at different points, usually with different data. This is where you can use “user DEFINed FuNctions”, “SUBroutines” and “PROCedures”. A user-defined function is really just a formula for doing one specific calculation. It may be one line long like

```
DEF FNcosh (X) = 0.5 * (EXP (X) + EXP (-X) )
```

or

```
DEF FNR (X) = X + 1/X
```

or may need many lines. The function can return a number, a string or even a truth value.

A user-defined function taking logical values:

FN: EXAMPLE 1

```
10 FOR X = 1 TO 10
20 IF FNA (X) PRINT X
30 NEXT
40 STOP
50 DEF FNA (T)
60 IF ABS (5 - T) < 2 = TRUE ELSE = FALSE
```

A user-defined function of two variables:

FN: EXAMPLE 2

```
10 FOR I = 1 TO 10
20 PRINT FNB (I, 5)
30 NEXT
40 STOP
50 DEF FNB (A, B)
60 IF A < B = B ELSE = A
```

A string valued user-defined function:

FN: EXAMPLE 3

```
10 FOR I = 1 TO 10
20 PRINT I, SPC (2); FNC (I)
30 NEXT
40 STOP
50 DEF FNC (T)
60 IF ABS (T) < 5 = "BIG" ELSE = "SMALL"
```

If the calculation is complex, the function may need to use its own variables to simplify or aid the calculation. It is usually important not to confuse these variables with others of the same name elsewhere in the program.

The LOCAL statement declares that the variables mentioned have no relation to any others elsewhere of the same name. They are only temporary and they will be lost at the end of that calculation. As a rule, all variables introduced in a DEF FN should be LOCAL.

When the computer meets the string " $= \text{FNalpha} (A, B, i)$ " as part of some line of program, it pauses and goes off to look for a corresponding DEF FNalpha statement. (If it has used FNalpha before, it will know where to look.) It then substitutes the values of A, B, i for the three parameters required by FNalpha, and starts to run through this DEF FNalpha piece of program until it meets " $= (\text{some expression})$ " which it takes to be the end of the calculation.

Often it is a process that you need to perform repeatedly, rather than a calculation. You do this with subroutines and/or procedures. These differ in a subtle way. Although most BASICS use subroutines, few have the use of procedures. A subroutine is simply a piece of program to which the computer is directed by a GOSUB statement. When it sees GOSUB 100, the first thing it does is to make a note of the line number of the very next line of program, placing this number on a stack. Then the computer goes off to line 100 and follows through the piece of program there until it meets a RETURN statement. RETURN to where?—to where it came from. It now takes the top address of the stack, and RETURNS as ordered. A subroutine may call other subroutines. Each call places the next line number on the stack, and each RETURN lifts one off. This explains why:

1. There is a limit to the number of subroutines that can call other subroutines: the stack has only got room for 26.
 2. Errors can occur if the computer meets a RETURN without a GOSUB or GOSUB without a RETURN.
-

Try to get that fly with your

FLYSWATTER

```
10 GOTO 50
20 I = (R = 1) * (P > 1) - (R = 2) * (P < 39)
30 J = (R = 4) * (Q > 1) - (R = 3) * (Q < 23)
40 PRINTTAB (P, Q); $$: RETURN
50 MODE 7
60 *FX 4, 1
70 F$ = "F": H$ = CHR$255: S$ = "□"
```

```

80  A = 0: B = 23: X = RND (39): Y = RND (23)
90  PRINTTAB (X, Y); F$; TAB (A, B); H$;
100 REPEAT
110  G = INKEY (10) - 135
120  R = RND (4)
130  P = X: Q = Y: GOSUB 20: X = X - I: Y = Y - J
140  PRINTTAB (X, Y); F$;
150  R = G: P = A: Q = B: GOSUB 20: A = A - I: B = B - J
160  PRINTTAB (A, B); H$
170  UNTIL A = X AND B = Y
180  STOP

```

Program notes

1. The subroutine (lines 20 to 40) determines if a point is to move, taking care if it is near the edge of the screen. This routine is applied to control both the fly's position (line 130) and the swatter's position (line 150).
2. Put in a timer or counter to see how long it takes to get the fly, and add some sound effects.

When told to GOSUB 1000, the computer starts to look for line 1000. It starts searching from the first line of program until it finds line 1000. It does this every time GOSUB 1000 is mentioned, without remembering where it was from the last time. For efficient use of subroutines then, put them at low line numbers.

A procedure is also a self contained piece of program to which the computer can be sent, simply by the command PROC. When meeting PROCAlbert (a, n) the computer looks for a DEF PROCAlbert line. If it has met PROCAlbert before, it will know where to look, as with a FN statement, but not like a GOSUB. It will then run through this piece of program until sent off somewhere else or until it meets an ENDPROC when, as with a subroutine, it returns to the line after it was invoked.

Procedures can also use LOCAL variables. This means that a procedure really is a self contained piece of program. As such it can be tried and tested, and can be used unchanged in many different programs. It can be used to make programs "structured" avoiding GOTOS and GOSUB. For example, a program which works out your bio-rhythms might look like this:

```

10  PROCInstructions
20  PROCtodaydate
30  PROCbirthdate
40  PROCcalculate
50  PROCdisplay
60  STOP

```

It is clear what the program intends to do, even though we haven't yet written the procedures. Here is one place where using meaningful names can be very helpful. Of course both PROCtodaydate and PROCbirthdate will expect the date in some form, so both of these might make use of a PROCdate procedure:

```

70  DEF PROCtodaydate
80  CLS

```

```

90 PRINT ' "type in today's date"
100 PROCdate
110 ENDPROC

```

Here's an example of the program LINESCOPE, where the plotting part of the program has been written using a procedure PROCPL.

LINESCOPE 2

```

10 MODE 5
20 VDU 29, 640; 512;
30 MX = 640: MY = 512
40 ST = RND (1000)
50 X = RND (1) * MX: Y = RND (1) * MY
60 SW = TRUE: C = 1
70 REPEAT
80 GCOL0, C
90 D = ST * (RND (MX/ST) - 1)
100 X1 = X: Y1 = Y: IF SW X1 = D ELSE Y1 = D
110 PROCPL (X, Y): PROCPL (-X, Y)
120 PROCPL (-X, -Y): PROCPL (X, -Y)
130 X = X1: Y = Y1
140 SW = NOT SW
150 C = (C + 1) MOD 4
160 UNTIL INKEY (0) > -1: RUN
170 DEFPROCPL (A, B) MOVE A, B
180 SA = SGN (A): SB = SGN (B)
190 PLOT 0, ST * (X1 = X) * SA, ST * (Y1 = Y) * SB
200 PLOT 85, A + (X1 - X) * SA, B + (Y1 - Y) * SB
210 ENDPROC

```

Program notes

1. The procedure PROCPL is used 4 times in the program, at lines 110 and 120, to produce the symmetrical pattern.
2. The procedure is defined in lines 170 to 210. Try changing this definition while leaving lines 10 to 160 the same, so that you still get a symmetrical pattern, but of a different type.
3. Notice the use of logical values in lines 190, 200.

23 Computing III

Time and space are the two precious commodities for the computer user. Each line of program and each variable you use consumes your computer memory, to which there is a limit. As a rough guide, the number of bytes of memory your program will need is:

(number of characters typed) + 5 * (number of variables)

Your program starts at PAGE, usually &E000 = 3584, and you can use from here up to HIMEM. The value of HIMEM is set by default according to which MODE you are using, but can be changed at the expense of using graphics. With a model A machine in MODE 4, you have only 2560 bytes available, and this you can easily use up. In order to minimize the amount of space you use try to:

1. Avoid unnecessary REMs.
2. Avoid spaces in the program.
3. Use short variable names.
4. Use integer variables where possible.
5. Use the resident integer variables A% to Z%.
6. Use procedures and subroutines to avoid repetition of lines.
7. Use as few variable names as possible, reviving any dead ones.

Fast as the computer may be, inefficient programs can slow it down noticeably. Games usually need to run as fast as possible. A bouncing ball program which took several hours for each move would soon lose its appeal. When thinking about time, don't forget your time. Is it worth an hour of your time to alter a program to run in 5 seconds instead of 7 seconds? In general the answer is no, of course. But it is a good idea to develop time-saving habits which become automatic and it is worth knowing where the time goes.

For example, compare the time taken to calculate with integer variables instead of FP variables, and the time saved by using an integer variable to control a FOR . . . NEXT loop.

TIME TEST

```
10 N% = 5000
20 A% = 6789
30 B% = 134
40 TIME = 0
50 FOR I% = 1 TO N%
```

60 C% = A% + B%

70 NEXT

80 PRINT TIME

Program notes

1. The calculation $C\% = A\% + B\%$ is performed over and over again, N% times, in a loop. This takes 4.23 seconds. Changing I% to I in line 50 increases this time to 5.84 seconds.
2. Try replacing line 50 with other calculations, and compare the times taken.

When the computer reads the name of a variable in some program line, it examines the first letter, then scans down a directory of names starting with this letter. If this one is in the directory, it reads off the address to find out where it has stored this variable, just like trying to find someone's telephone number in a telephone directory. If the variable's name is not on the list then its value is added on to the bottom of the list. The computer looks in its memory for some space to store it, then writes this address in the directory, beside the name. This applies to each of the 52 possible starting letters. In the same way, a single directory is written for the procedures, and another for the user-defined functions.

This means that the variables are added to the list in the order in which they are encountered during the RUN. It usually makes little difference what this order is in terms of time. But if X is a variable constantly being used in the program then put $X = 0$ in the first line of program, to get it known. Then it'll be top of the list.

Your program occupies a piece of memory from PAGE up to LOMEM. Usually $PAGE = \&E00 = 3584$ while LOMEM varies with your program. The variables are stored from TOP upwards, and usually $TOP = LOMEM$. Each MODE causes HIMEM to be set at some level beyond which you cannot go (unless you change HIMEM). The memory beyond HIMEM is used for the screen display. In MODE 7 the screen can only be used for text and there are $40 * 25 = 1000$ possible character positions, so only 1000 bytes are needed for the screen display. In MODE 4 with a resolution of $320 \times 256 = 81920$ points, you need $81920/8 = 10240$ bytes in order to be able to refer to any point.

The contents of any byte can be accessed by PEEKing at its address. If X is the address of a byte, then ?X is the value of this byte. This number can be used in a program:

IF ?X = 140 THEN STOP

For example in MODE 7, the byte with address $HIMEM + 40 * Y + X$ is the position (X, Y) on the screen, so that $?(HIMEM + 40 * Y + X)$ contains the ASCII code for the character at this position.

Reach home as quickly as you can going through the

MAZE

10 MODE 7

20 *FX 4, 1

30 HX = 39: HY = 21: X0 = 0: Y0 = 1

40 S\$ = "□": B\$ = "*"

50 PRINTTAB (HX, HY) "H"

60 A\$ = STRING\$(40, CHR\$(255))

70 FOR Y = 2 TO 20 STEP 2

```

80 PRINTTAB (0, Y) A$
90 PRINTTAB (RND (39), Y); S$
100 NEXT
110 PRINTTAB (X0, Y0) B$;
120 TIME = 0: REPEAT
130 G = GET - 135
140 X1 = X0 - (G = 1) * (X0 > 0) + (G = 2) * (X0 < 39)
150 Y1 = Y0 - (G = 4) * (Y0 > 0) + (G = 3) * (Y0 < 22)
160 IF? (HIMEM + X1 + 40 * Y1) = 255 THEN 130
170 PRINTTAB (X0, Y0) S$
180 X0 = X1: Y0 = Y1
190 PRINTTAB (X0, Y0) B$
200 UNTIL X0 = HX AND Y0 = HY
210 PRINTTAB (0, 21); "HOME IN □";
220 PRINT TIME DIV 100; "□ SECONDS"
230 *FX 4, 0

```

Program notes

1. CHR\$ (255) is a blank square. The loop 70 to 100 prints ten horizontal walls, each with a single gap in some random position.
2. Line 160 checks to see if there is a wall in the way.

Produce large letters on the screen with

SUPERSTAR

```

10 INPUT "Type character", A$
20 MODE 4: PRINTA$
30 FOR I = 0 TO 7
40 A = ? (HIMEM + I)
50 PRINTTAB (10); FNBIG (A)
60 NEXT
70 RUN
80 DEFFNBIG (A)
90 A$ = " "
100 FOR J = 1 TO 8
110 Q = A MOD 2: A = A DIV 2
120 IF Q = 0 A$ = "□" + A$ ELSE A$ = "*" + A$
130 NEXT
140 = A$

```

Program notes

1. The character which was input is printed (in line 20) in the top left hand corner of the screen. Lines 30 to 60 scan this part of the screen/memory and the shape of this character is picked up and converted (by means of the user-defined function FNBIG) into a string of stars which are then displayed. The eight strings fit together to give a large version of this character.
 2. If you combine this program with PRINT CHR\$(141) in the right places, you'll get even bigger letters!
-

Certain addresses contain key information about the computer's operations. For example the bytes 2 and 3 contain the address of the end of the variable storage area of your program, so $256 * (?3) + ?2$ is the end of your variables. Bytes &482, &483 contain the address of the first variable starting with letter A, if there is one. If $256 * (? &483) + (? &482) = 0$ then there is currently no variable in use whose name starts with the letter A. The contents of any address in RAM can be altered by POKEing. If you want to put the value 143 into memory address X then $?X = 143$ does the trick.

24 BASIC VI

There are 26 resident integer variables, called A% to Z%. These live in a special part of the memory (on page 4) and are unaltered by RUN, CLEAR or BREAK, so can be used to transfer information from one program to another, or one RUN to the next. To find their values at any time, try the following program.

RESIDENT INTEGER VALUES

```
10 J = &4000
20 FOR I = 65 TO 90
30 J = J + 4
40 IF COUNT > 20 PRINT
50 PRINT SPC (2) CHR$(I); "% = ";
60 PRINT!J;
70 NEXT
80 PRINT
```

Program note

Each resident integer variable occupies 4 consecutive bytes starting from &4000, the home of @ %. If N is an address then !N is the integer represented by the 4 bytes starting at N (see line 60).

25 PRINT V

Each character printed on the screen is made up of small dots in an 8×8 matrix. Each row consists of 8 dots, each of which can either be on or off, so can be thought of as an 8 bit binary word, i.e. as a byte. The 8 rows produce 8 bytes, so the means of displaying each graphic character is contained in 8 bytes.

When asked, for example, to PRINTCHR\$(65), the computer scans its memory looking for the 8 bytes which it knows make up the shape of CHR\$(65), and on finding them puts them on the screen to get the familiar shape "A" we know and love.

We can use this method to construct our own characters using VDU 23. (This has no effect in MODE 7.) To use this, draw a shape on an 8×8 matrix, and calculate the row numbers say A1, A2 . . . A8. Any ASCII character can be reprogrammed to display this shape. Characters numbered 224 to 255 have been conveniently reserved for this purpose so you can still use the old ones when you want. The command VDU 23, 224, A1, A2, A3, A4, A5, A6, A7, A8 will cause the computer to print your shape in response to the PRINTCHR\$(224) command.

DESIGN PRINT

```
10 MODE 7
20 DIM A (8)
30 PRINT ' ' SPC (11); "12345678"
40 FOR I = 1 TO 8
50 PRINT I;
60 INPUT LINE A$
70 L = LEN (A$): IF L > 8 THEN L = 8
80 A (I) = FNA (A$)
90 NEXT
100 VDU 23, 224, A(1), A(2), A(3), A(4), A(5), A(6), A(7), A(8)
110 MODE 6
120 FOR I = 1 TO 20
130 PRINT STRING$ (I, CHR$224): NEXT
140 STOP
150 DEF FNA (A$)
160 B = 0: F = 2 ↑ (8 - L)
```

```

170 FOR J = L TO 1 STEP -1
180 IF MID$ (A$, J, 1) < > "□" THEN B = B + F
190 F = 2 * F
200 NEXT
210 = B

```

Program notes

1. Type in a shape made up of stars and spaces, one row at a time. Each row is input using INPUTLINE (line 60) which takes the line literally, including leading and trailing spaces if any.
2. In line 100, this shape is transferred to character 224, its parameters having been calculated by the user-defined function FNA.

Additional Programs

Display the calendar for any month with

CALENDAR MONTH

```
10 @ % = 4
20 DATA 1, -2, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1
30 DATA S, M, T, W, T, F, S
40 INPUT "MONTH ", M, "YEAR ", Y
50 IF Y > 99 THEN Y = Y - 1900
60 Z = INT (365.25 * (Y - 1) )
70 LEAP = 0: IF Y MOD 4 < > 0 GOTO 110
80 IF Y MOD 100 < > 0 THEN LEAP = 1
90 IF Y MOD 400 = 0 THEN LEAP = 1
100 RESTORE 20
110 FOR I = 1 TO M
120 READ N: Z = Z + N + 30: NEXT
130 N = 30 + N: Z = Z - N + 2
140 IF LEAP = 1 AND M = 2 THEN N = 29
150 IF M > 2 THEN Z = Z + LEAP
160 B = 1 - (Z MOD 7)
170 RESTORE 30: PRINT ' '
180 FOR I = 1 TO 7
190 READ A$: PRINT A$; SPC (2);
200 FOR J = 1 TO 6
210 IF B < 1 OR B > N PRINT SPC (4);
220 IF 0 < B AND B <= N PRINT B;
230 B = B + 7: NEXT: PRINT
240 B = B - 41: NEXT
```

Program notes

1. This program just about fits on to the screen and is ripe for improvement to give a better display of the calendar.
2. The number of days above 30 in each month is stored in the DATA statement in line 20.
3. The calculations in lines 50 to 160 determine the day of the first of the month.

If you have artistic tendencies, see how you get on with

SKETCH

```
10 MODE 4
20 *FX 4, 1
30 *FX 12, 5
40 *FX 11, 1
```

```

50 VDU 23;8202;0;0;0;
60 VDU 29,640;512;
70 X = 0; Y = 0
80 MOVE X, Y
90 REPEAT
100 G = GET - 135
110 X = X + 8 * ( (G = 1) - (G = 2) )
120 Y = Y + 8 * ( (G = 3) - (G = 4) )
130 DRAW X, Y
140 UNTIL G = -103
150 MODE 7
160 *FX 4, 0
170 *FX 11, 0
180 *FX 12, 0
190 VDU 23;29194;0;0;0;
200 STOP

```

Program notes

1. The delay and repeat rate of the keyboard is changed in lines 30 and 40 and reset in lines 170 and 180.
2. The editing keys are used to control the graphics cursor, which draws as it moves.
3. The value 8 in lines 110 and 120 controls the fineness of the drawing.
4. Drawing stops when the space bar is touched (see line 140).

A hidden word game:

SPOT MY WORD

```

10 READ A$: IF A$ = "****" THEN STOP
20 MODE 5
30 VDU 19, 1, 7, 0, 0, 0
40 VDU 19, 2, 7, 0, 0, 0
50 NL = 10: Y = RND (NL - 1): X = RND (19 - LEN A$)
60 R = 20 * Y + X
70 COLOUR 1
80 PRINT ' ' "□ SPOT □"; A$ ' '
90 FOR I = 1 TO R
100 PRINT CHR$ (64 + RND (26) ); : NEXT
110 COLOUR 2: PRINT A$;
120 COLOUR 1
130 FOR I = R + LEN A$ + 1 TO 20 * NL
140 PRINT CHR$ (64 + RND (26) ); : NEXT
150 TIME = 0

```

```

160 REPEAT UNTIL GET = 32
170 VDU 19, 1, 1, 0, 0, 0
180 COLOUR 2
190 PRINT "TIME "; TIME DIV 100
200 REPEAT UNTIL GET = 13
210 GOTO 10
220 DATA CAT, DOG, HOUSE, OSTRICH, ****

```

Program note

A word is read from a DATA statement in line 10. Colours 1 and 2 are both set to white in lines 30 and 40, and the chosen word is printed in colour 2 among random letters printed in colour 1. But because both colours are white, it is hidden. When the space bar is hit, the colours revert to their default values (line 170) making the hidden word stand out.

Deal yourself a hand of

CARDS

```

10 A$ = "diamondsclubs □ □ □"
20 A$ = A$ + "hearts □ □ spades □ □"
30 MODE 7
40 DIM A%(52)
50 FOR I% = 1 TO 52
60 A%(I%) = I%: NEXT
70 INPUT "How many cards ", N%
80 FOR J% = 52 TO 2 STEP -1
90 R% = RND(J%)
100 T% = A%(R%): A%(R%) = A%(J%): A%(J%) = T%
110 NEXT
120 FOR K% = 52 TO 53 - N% STEP -1
130 B% = A%(K%)
140 S% = B% MOD 4
150 F% = 1 + B% MOD 13
160 PRINT F%; SPC(2); "□ of □";
170 PRINT MID$(A$, 8 * S% + 1, 8)
180 NEXT

```

Program notes

1. Lines 40 to 60 initialize the pack, while lines 80 to 110 shuffle the entire pack. After this loop, the array A% contains the numbers from 1 to 52 in some order.
2. The cards are converted into suits and face values in lines 130 to 150, just like in ONE CARD.

Watch a knight moving at random on a chessboard with

KNIGHT'S MOVE

```
10  MODE 7
20  VDU 28, 16, 16, 24, 8
30  FOR I = 0 TO 7: FOR J = 0 TO 7
40  K = (I + J) MOD 2
50  IF K PRINT "□"; ELSE PRINT "*";
60  NEXT: PRINT: NEXT
70  I = 0: J = 0
80  REPEAT
90  F = RND (2): S = 3 - F
100 IF RND (2) = 1 S = -S
110 IF RND (2) = 1 F = -F
120 I1 = I + F: J1 = J + S
130 IF ABS (3.5 - I1) > 4 THEN 90
140 IF ABS (3.5 - J1) > 4 THEN 90
150 K = (I + J) MOD 2: PRINTTAB (I, J);
160 IF K PRINT "□"; ELSE PRINT "*";
170 I = I1: J = J1
180 PRINTTAB (I, J); "K";
190 REPEAT UNTIL GET = 32
200 UNTIL FALSE
```

Program notes

1. Starting from any point on a chessboard, a knight can eventually reach any square. In this program a knight moves at random. Each move is determined in lines 90 to 120. If the move would go off the board, then this is detected in lines 130 and 140 and ignored.
2. Line 190 gives you time to see what is happening: the knight moves when you hit the space bar.

Each time this runs, you'll see a different

KALEIDOSCOPE

```
10  DIM X (3), Y (3), C (6), S (6)
20  DIM X1 (3, 12), Y1 (3, 12)
30  FOR J = 1 TO 6
40  C (J) = COS (J * PI/3): S (J) = SIN (J * PI/3)
50  NEXT: MX = 640: A = SIN (PI/6): CC = 1
60  MODE 5: VDU 29, 640, 512;
70  REPEAT: GCOL0, CC
80  FOR I = 1 TO 3
```

```

90  X(I) = 50 + MX / 60 * RND(60)
100 Y(I) = RND(1) * A * X(I): NEXT
110 FOR CT = 0 TO 1: FOR I = 1 TO 6
120  C = C(I): S = S(I): FOR J = 1 TO 3
130  X1(J, I + 6 * CT) = X(J) * C + Y(J) * S
140  Y1(J, I + 6 * CT) = -X(J) * S + Y(J) * C
150 NEXT: NEXT
160 FOR K = 1 TO 3
170  X(K) = -X(K): NEXT: NEXT
180 FOR I = 1 TO 12
190  MOVE X1(1, I), Y1(1, I)
200  MOVE X1(2, I), Y1(2, I)
210  PLOT 85, X1(3, I), Y1(3, I): NEXT
220  CC = CC + 1: IF CC = 4 THEN CC = 0
230  UNTIL INKEY(0) > -1: RUN

```

Program note

This program draws a hexagonal kaleidoscope in the 4 default colours. The arrays store the necessary calculations, which help to speed up the display. Lines 90 and 100 produce a random triangle and lines 110 to 150 reflect this triangle to form twelve triangles. Lines 180 to 210 fill in these triangles with colour.

Think of a function then

PLOT IT

```

10  MODE 7
20  N = 100: DIM A(N)
30  INPUT " " "function to plot □", F$
40  INPUT "from X = □", A$: A = EVAL(A$)
50  INPUT "to X = □", B$: B = EVAL(B$)
60  D = B - A: IX = D/N
70  X = A: Y = EVAL(F$)
80  I = 0: MIN = Y: MAX = Y
90  FOR X = A TO B STEP IX
100 Y = EVAL(F$)
110 A(I) = Y
120 IF Y < MIN THEN MIN = Y
130 IF Y > MAX THEN MAX = Y
140 I = I + 1
150 NEXT
160 RANGE = MAX - MIN
170 SX = 1280/N: SY = 1024 / RANGE

```

```

180 MODE 5
190 MOVE 0, SY * (A (0) - MIN)
200 FOR I = 1 TO N
210 DRAW SX * I, SY * (A (I) - MIN): NEXT

```

Program notes

1. You input a function of X as a string in line 30, with the range of values to be plotted in lines 40 and 50. The function is evaluated at N points in this range, in this case N = 100 (line 20).
 2. The maximum and minimum values of this function are calculated (lines 120 and 130) and used to scale the graph before plotting so that it fits on to the screen (line 170).
 3. Notice that this program uses the three variables A, A\$ and A (N) which are not to be confused.
-

Use the sieve of Eratosthenes to find those

PRIMES

```

10 MODE 7
20 @ % = 5
30 N% = 100
40 D% = 2 * N% + 3
50 DIM A% (N%)
60 PRINT "Primes less than "; D%
70 FOR I = 2 TO 3: PRINT I; : NEXT
80 J% = 0: T% = 3
90 REPEAT
100 FOR L% = T% TO D% STEP 2 * T%
110 A% ((L% - 3) DIV 2) = 1
120 NEXT
130 FOR I% = J% TO N%
140 SW% = 0
150 IF A% (I%) = 1 THEN 190
160 SW% = 1
170 J% = I%: T% = 2 * I% + 3: I% = N%
180 PRINT T%;
190 NEXT
200 UNTIL SW% = 0

```

Program note

You need a mathematical bent to understand this program. However you can still run it without knowing how it works. What it does is to display all the prime numbers up to D%, which in this case is 203. If you increase N% in line 30 then it's worth also adding lines between 180 and 190 to make the program pause when the screen is full, otherwise the numbers will scroll away before you can look at them.

An example of the bubble sort is

ALPHABETIC SORT

```
10 DIM A (20)
20 INPUT ' ', A$
30 L = LEN (A$)
40 FOR I = 1 TO L
50 A (I) = ASC (RIGHT$ (A$, I) )
60 NEXT
70 N1 = L
80 N1 = N1 - 1: UNSORTED = FALSE
90 FOR I = 1 TO N1
100 Q = A (I + 1): R = A(I)
110 IF Q >= R THEN 140
120 UNSORTED = TRUE
130 A (I + 1) = R: A (I) = Q
140 NEXT
150 IF UNSORTED GOTO 80
160 FOR I = 1 TO L
170 PRINT CHR$ (A (I) );
180 NEXT: PRINT
200 RUN
```

Program notes

1. A string is read in and the ASCII codes of its characters stored in the array A (lines 20 to 60).
 2. A bubble sort of the entries in A takes place between lines 70 and 150, with the logical variable "UNSORTED" used to see if the array is already sorted.
 3. Lines 160 to 180 print the letters of A\$ in alphabetical order.
 4. You could combine this with ANAGRAM to make a fiendish game.
 5. If you took each word in the dictionary and put it through this program and then made a new dictionary with the words that came out, which would be the first three letter word in the dictionary? What about 4 and 5 letter words?
-

POWERS OF TWO

```
10 N = 10
20 E% = 4: K% = 10 ↑ E%
30 DIM A% (N)
40 A% (0) = 1
50 FOR I% = 1 TO 40
60 C% = 0
70 FOR J% = 0 TO N
80 X% = A% (J%) * 2 + C%
```

```

90  A% (J%) = X% MOD K%
100 C% = X% DIV K%
110 NEXT
120 SW = TRUE
130 PRINT "2 ↑"; I%; "□ = □";
140 FOR L = N TO 0 STEP -1
150  X% = A% (L): X$ = STR$ (X%): S = LEN X$
160  IF SW AND X% = 0 THEN 200
170  IF SW AND X% > 0 SW = FALSE: GOTO 190
180  PRINT STRING$ (E% - S, "0");
190  PRINT X$;
200 NEXT
210 PRINT
220 NEXT

```

Program notes

1. This is a simple example of how you can get past the limitations of size by using arrays. The powers of 2 up to the 40th are calculated by the usual "long multiplication" method taught at school. The array A% stores the results of the calculations, and has room for 4 * N digits. In this case N is 10, set in line 10, so that the program is able to calculate all powers of 2 which require no more than 40 digits.
2. If you replace the 2 in line 80 by I%, the program will produce I%! the factorial of I%. Don't forget to change line 130, though.
3. Having made the calculations and stored the results in A%, the computer displays the answer by means of lines 140 to 200. The logical variable SW is used to suppress leading zeros.

A crude but simple searching program:

KEYSEARCH

```

10  INPUT "How many keywords", N
20  DIM A$ (N)
30  PRINT "Type in □"; N; "□ keywords:"
40  FOR I = 1 TO N
50  PRINT I; :INPUT A$ (I): NEXT
60  INPUT "How many fields/record", F
70  INPUT "Search which field □", K
80  DIM B$ (F)
90  PRINT "TYPE IN RECORDS"
100 PRINT "Next record:"
110 FOR I = 1 TO F
120 PRINT I;
130 INPUT B$ (I): NEXT

```

```

140 C$ = B$ (K)
150 FOUND = FALSE
160 FOR J = 1 TO N
170 A$ = A$ (J): L = LEN (A$)
180 IF L > LEN C$ THEN 210
190 IF INSTR (C$, A$) = 0 THEN 210
200 FOUND = TRUE: J = N
210 NEXT
220 IF FOUND PRINT "****" A$
230 GOTO 100

```

Program notes

1. Suppose you have a file of records, say on cassette. For example, each record might consist of 4 fields: name, address, county, telephone number. You need to search through these records looking for those which fulfil some condition. You may want the records of everyone who lives in Wales, or whose name begins with the letter T.
This program asks you to supply a list of keywords. These are the words the program will look out for. It then asks you how many fields make up each record, and then, the important question, which field it is to examine. In the case of a record consisting of name, address, county and telephone number you would specify field 3 if you want to search the counties, but field 1 if it's names you're interested in.
2. In this example the records are typed in (lines 90 to 130). Of course this is a bit pointless, but the program could easily be adapted to receive this data from a cassette, in which case having answered the questions you just sit back and await a display of the records you require.

Modify your program with a

DATA CHANGE

```

10 GOTO 150
20 A = ? (S + 3) - 1
30 IF ? (S + 6) <> 43 RETURN
40 K = 1
50 FOR J = S + 6 TO S + A
60 ?J = ASCMID$(A$, K, 1)
70 K = K + 1
80 NEXT
90 SW = TRUE
100 RETURN
110 DATA + + + + + + + + +
120 DATA + + + + + + + + +
130 DATA + + + + + + + + +
140 DATA + + + + + + + + +
150 SW = FALSE

```

```

160 INPUT A$: L = LEN A$: IF L > 10 L = 10
170 S = PAGE
180 T = ? (S + 5)
190 IF T = &DC GOSUB 20
200 IF SW THEN STOP
210 A = ? (S + 3): IF A = 0 STOP
220 S = S + A: GOTO 180

```

Program notes

1. This program reads a string of up to 10 characters (line 160) and looks for a suitable DATA statement in which to store it.
2. Lines 170 to 210 scan the memory looking for DATA statements. If one is found (line 190) and it is "unused" then the string is POKed in to it in lines 40 to 80.
3. There *must* be one space between each line number and DATA statement in your program listing.

Program Index

Algebra	48	Knight's move	92
Alphabetic sort	95	Know your tables	26
Anagram	67	Linescope 1	12
Another window	74	Linescope 2	79
Back to one	5	Linewidth	41
Binary	57	Maze	81
Blastoff	38	Move a string	53
Bounce	22	Number storage	60
Calendar month	89	One card	44
Capitals	24	On error example	55
Cards	91	Password	45
Chemical formulae	75	Phonelist	48
Chessboard	42	Plot it	93
Circle	10	POS example	63
Code	45	Powers of two	95
Colour pairs	18	Primes	94
Contours	16	Random walk	11
Conversion	66	Reaction	25
Data change	97	Reciprocal	59
Dayfinder!	30	Rectangle	9
Daynumber	7	Resident integer values	84
Diary	28	Road	28
Dice roll	32	Rounding errors: example 1	35
Design print	85	Rounding errors: example 2	35
Doubles	6	Rounding errors: example 3	36
Duck shoot	71	Shooting star	26
Easter	7	Siren	37
EOR table	61	Sketch	89
Factorization	47	Spiral	39
Field width 1	22	Spot my word	90
Field width 2	51	Stars and stripes	15
Flash	19	Star symmetry	23
Flower	13	Superstar	82
Flyswatter	77	Symmetry	17
FN: example 1	76	Teletext shapes	73
FN: example 2	76	Teletext test	71
FN: example 3	77	Thread pattern	11
Hangman	68	Time test	80
Happy birthday tune	65	Trace example	56
Happy birthday verse	31	Two dice	66
Heads or tails	5	Upper to lower	62
Hexadecimal	58	VPOS example	63
How far?	31	Weave	17
Interference patterns	10	Wheels within wheels	13
Kaleidoscope	92	Window	74
Keysearch	96		

Other titles of interest

Easy Programming for the BBC Micro
Eric Deeson

PEEK, POKE, BYTE & RAM! Basic Programming for the ZX81
Ian Stewart & Robin Jones

'Far and away the best book for ZX81 users new to computing'—*Popular Computing Weekly*

'... the best introduction to using this trail-blazing micro'—*Computers in Schools*

'One of fifty books already published on the Sinclair micros, it is the best introduction accessible to all computing novices'—*Laboratory Equipment Digest*

Machine Code and better Basic
Ian Stewart & Robin Jones

The ZX81 Add-On Book
Martin Wren-Hilton

Easy Programming for the ZX Spectrum
Ian Stewart & Robin Jones

ComputerPuzzles: For Spectrum and ZX81
Ian Stewart & Robin Jones

Games to Play on Your ZX Spectrum
Martin Wren-Hilton

Available from January '83

Spectrum in Education
Eric Deeson

Further Programming for the ZX Spectrum
Ian Stewart & Robin Jones

Spectrum Machine Code
Ian Stewart & Robin Jones

Brainteasers for BASIC Computers
Gordon Lee

Shiva Software

Spectrum Special 1
Ian Stewart & Robin Jones

A selection of 10 educational games and puzzles.

Plus more to come!

Following on from EASY PROGRAMMING FOR THE BBC MICRO this book takes a program-based approach to further your appreciation of the computer.

Over 90 programs are listed in the text, each of which is designed to show one of the special features of the BBC Micro. The listings are accompanied by notes which pick out particular points of interest and give hints on how the programs can be extended and improved.

A section of slightly longer "Additional Programs" is included at the end of the book which can be copied and RUN at any time.

Programs include:

- Duck Shoot
- Kaleidoscope
- Flyswatter
- Maze
- Random Walk

Further your programming skills
— Be a Better Computer programmer!

